```
<VIDEO>Path            { return(PATH); }
<VIDEO>Files     { return(FILES); }
<VIDEO>Transform       { return(TRANSFORM); }
<VIDEO>None      { return(TRANSFORM_NONE); }
<VIDEO>Wavelet   { return(TRANSFORM_WAVE); }
<VIDEO>Start     { return(START); }
<VIDEO>End             { return(END); }
<VIDEO>Length    { return(LEN); }
<VIDEO>Dimensions      { return(DIM); }
<VIDEO>Header    { return(HEADER); }
<VIDEO>Offsets   { return(OFFSETS); }
<VIDEO>Size            { return(SIZE); }
<VIDEO>Precision       { return(PRECISION); }
<VIDEO>Yes                     { yylval.bool=True; return(BOOLEAN); }
<VIDEO>No                      { yylval.bool=False; return(BOOLEAN); }

<BATCH>Load                    { return(LOAD); }
<BATCH>Save                    { return(SAVE); }
<BATCH>SaveAbekus      { return(SAVE_ABEKUS); }
<BATCH>Compare                 { return(COMPARE); }
<BATCH>Drop                    { return(DROP); }
<BATCH>ImportKLICS     { return(IMPORT_KLICS); }
<BATCH>Transform       { BEGIN BATCH_TRANS; return(TRANSFORM); }
<BATCH>Compress                { BEGIN BATCH_COMP; return(COMPRESS); }
<BATCH>Xwave           { return(XWAVE); }
<BATCH>Shell           { return(SHELL); }
<BATCH>Copy                    { return(COPY); }
<BATCH>Direct          { return(DIRECT_COPY); }
<BATCH>Diff                    { return(DIFF); }
<BATCH>LPFzero                 { return(LPF_WIPE); }
<BATCH>LPFonly                 { return(LPF_ONLY); }
<BATCH>RGB-YUV                 { return(RGB_YUV); }
```

```
<BATCH>Gamma           { return(GAMMA); }

<BATCH_COMP>VideoName     { return(VIDEO_NAME); }
<BATCH_COMP>Stats         { return(STATS_NAME); }
<BATCH_COMP>Binary        { return(BIN_NAME); }
<BATCH_COMP>Yes                  { yylval.bool=True; return(BOOLEAN); }
<BATCH_COMP>No                   { yylval.bool=False; return(BOOLEAN); }
<BATCH_COMP>Still         { return(STILL_MODE); }
<BATCH_COMP>Video         { return(VIDEO_MODE); }
<BATCH_COMP>AutoQuant     { return(AUTO_Q); }
<BATCH_COMP>QuantConst    { return(QUANT_CONST); }
<BATCH_COMP>ThreshConst   { return(THRESH_CONST); }
<BATCH_COMP>BaseFactor    { return(BASE_FACTOR); }
<BATCH_COMP>DiagFactor    { return(DIAG_FACTOR); }
<BATCH_COMP>ChromeFactor  { return(CHROME_FACTOR); }
<BATCH_COMP>Decision      { return(DECISION); }
<BATCH_COMP>Feedback      { return(FEEDBACK); }
<BATCH_COMP>Maximum              { return(DEC_MAX); }
<BATCH_COMP>SigmaAbs      { return(DEC_SIGABS); }
<BATCH_COMP>SigmaSqr      { return(DEC_SIGSQR); }
<BATCH_COMP>Filter        { return(FILTER); }
<BATCH_COMP>None          { return(FLT_NONE); }
<BATCH_COMP>Exp                  { return(FLT_EXP); }
<BATCH_COMP>CmpConst      { return(CMP_CONST); }
<BATCH_COMP>FrameRate     { return(FPS); }
<BATCH_COMP>Bitrate       { return(BITRATE); }
<BATCH_COMP>Buffer        { return(BUFFER); }
<BATCH_COMP>\{                   { return(LEFT_BRACE); }
<BATCH_COMP>\}                   { END; BEGIN BATCH;
return(RIGHT_BRACE); }

<BATCH_TRANS>VideoName    { return(VIDEO_NAME); }
```

```
<BATCH_TRANS>Direction      { return(DIRECTION); }
<BATCH_TRANS>Space { return(SPACE); }
<BATCH_TRANS>Precision      { return(PRECISION); }
<BATCH_TRANS>Yes            { yylval.bool=True; return(BOOLEAN); }
<BATCH_TRANS>No                     { yylval.bool=False; return(BOOLEAN); }
<BATCH_TRANS>\{                     { return(LEFT_BRACE); }
<BATCH_TRANS>\}             { END; BEGIN BATCH; return(RIGHT_BRACE); }

[. \t\n]          { ; }

%%
yywrap() { return(1); }
```

source/Transform.h

```
typedef         struct  {
        Video src;
        char    name[STRLEN], src_name[STRLEN];
        int     space[2], precision;
        Boolean dirn;
} TransCtrlRec, *TransCtrl;
```

source/Video.h

```
typedef struct {
	char	names[4][STRLEN];
} AbekusCtrlRec, *AbekusCtrl;
```

source/makefile

```
# Xwave Makefile
#
CFLAGS = -O -I../include
LIBS   = -lXaw -lXmu -lXt -lXext -lX11 -lm -ll -L/usr/openwin/lib

.KEEP_STATE:
.SUFFIXES: .c .o

xwaveSRC = Select.c Convert.c xwave.c InitMain.c Pop2.c Video2.c Malloc.c
InitFrame.c \
		Frame.c Transform.c Convolve3.c Update.c Image.c Menu.c
PullRightMenu.c \
		NameButton.c SmeBSBpr.c Process.c Lex.c Gram.c Parse.c Color.c \
		Bits.c Storage.c Copy.c Message.c Palette.c ImportKlics.c Icon3.c Klics5.c
\
		KlicsSA.c KlicsTestSA.c ImportKlicsSA.c ImpKlicsTestSA.c

objDIR = ../$(ARCH)

xwaveOBJ = $(xwaveSRC:%.c=$(objDIR)/%.o)

$(objDIR)/xwave: $(xwaveOBJ)
	gcc -o $@ $(xwaveOBJ) $(LIBS) $(CFLAGS)
	echo ●●●●●●●●●●●●●●●●●●●●●●●

$(xwaveOBJ): $$(@F:.o=.c) ../include/xwave.h
	gcc -c $(@F:.o=.c) $(CFLAGS) -o $@

Lex.c: Gram.c Lex.l
```

```
        lex Lex.l
        mv lex.yy.c Lex.c

Gram.c: Gram.y
        bison -dlt Gram.y
        mv $(@F:.c=.tab.h) ../include/Gram.h
        mv $(@F:.c=.tab.c) Gram.c
```

include/Bits.h

```
#ifndef _Bits_h
#define _Bits_h

typedef         struct  {
        unsigned        char    buf;
        int     bufsize;
        FILE    *fp;
} BitsRec, *Bits;

#endif
```

include/DTheader.h

```
typedef struct DTheader {
   char file_id[8];          /* "DT-IMAGE" */
   char struct_id;           /* 1 */
      char prod_id;              /* 4 */
      char util_id;              /* 1 */
      char board_id;                    /* 2 */
      char create_time[9];  /* [0-1]year, [2]month, [3]dayofmonth, [4]dayofweek,
[5]hour, [6]min, [7]sec, [8]sec/100 */
      char mod_time[9];          /* as create_time */
      char datum;                       /* 1 */
      char datasize[4];          /* 1024?? */
      char file_struct;          /* 1 */
      char datatype;                    /* 1 */
      char compress;                    /* 0 */
      char store;                       /* 1 */
      char aspect[2];                   /* 4, 3 */
      char bpp;                         /* 8 */
      char spatial;              /* 1 */
      char width[2];                    /* 512 */
      char height[2];                   /* 512 */
      char full_width[2];        /* 512 */
      char full_height[2];  /* 512 */
      char unused1[45];
      char comment[160];
      char unused2[256];
} DTheader;
```

include/Icon.h

```
typedef         enum {
        FW_label, FW_icon, FW_command, FW_text, FW_button, FW_icon_button,
FW_view, FW_toggle,
        FW_yn,
        FW_up, FW_down, FW_integer,
        FW_scroll, FW_float,
        FW_form,
} FormWidgetType;

typedef         enum {
        SW_below, SW_over, SW_top, SW_menu,
} ShellWidgetType;

typedef         struct {
        String name;
        String contents;
        int             fromHoriz, fromVert;
        FormWidgetType     type;
        String hook;
} FormItem;
```

include/Image.h

```
/*
 * $XConsortium: Image.h,v 1.24 89/07/21 01:48:51 kit Exp $
 */

/***********************************************************
Copyright 1987, 1988 by Digital Equipment Corporation, Maynard, Massachusetts,
and the Massachusetts Institute of Technology, Cambridge, Massachusetts.

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the names of Digital or MIT not be
used in advertising or publicity pertaining to distribution of the
software without specific, written prior permission.

DIGITAL DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING
ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL
DIGITAL BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL
DAMAGES OR
ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR
PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER
TORTIOUS ACTION,
```

```
ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF
THIS
SOFTWARE.

******************************************************************/

#ifndef _XawImage_h
#define _XawImage_h

/***********************************************************************
 *
 * Image Widget
 *
 ***********************************************************************/

#include <X11/Xaw/Simple.h>
#include <X11/Xmu/Converters.h>

/* Resources:

 Name                Class              RepType         Default Value
 ----                -----              -------         -------------
 border              BorderColor        Pixel           XtDefaultForeground
 borderWidth         BorderWidth        Dimension       1
 cursor              Cursor             Cursor          None
 destroyCallback     Callback           XtCallbackList  NULL
 insensitiveBorder   Insensitive        Pixmap          Gray
 mappedWhenManaged   MappedWhenManaged  Boolean         True
 sensitive           Sensitive          Boolean         True
 bitmap              Bitmap             Pixmap          NULL
 callback            Callback           XtCallbackList  NULL
 x                   Position           Position        0
```

```
y                Position         Position      0

*/

#define XtNbitmap   "bitmap"

#define XtCBitmap   "Bitmap"

/* Class record constants */

extern WidgetClass imageWidgetClass;

typedef struct _ImageClassRec *ImageWidgetClass;
typedef struct _ImageRec      *ImageWidget;

#endif /* _XawImage_h */
/* DON'T ADD STUFF AFTER THIS #endif */
```

include/ImageHeader.h

```
/* Author: Philip R. Thompson
 * Address: phils@athena.mit.edu, 9-526
 * Note: size of header should be 1024 (1K) bytes.
 * $Header: ImageHeader.h,v 1.2 89/02/13 09:01:36 phils Locked $
 * $Date: 89/02/13 09:01:36 $
 * $Source: /mit/phils/utils/RCS/ImageHeader.h,v $
 */

#define IMAGE_VERSION    3

typedef struct ImageHeader {
    char file_version[8];   /* header version */
    char header_size[8];    /* Size of file header in bytes */
    char image_width[8];    /* Width of the raster image */
    char image_height[8];   /* Height of the raster imgage */
    char num_colors[8];     /* Actual number of entries in c_map */
    char num_channels[8];   /* 0 or 1 = pixmap, 3 = RG&B buffers */
    char num_pictures[8];   /* Number of pictures in file */
    char alpha_channel[4];  /* Alpha channel flag */
    char runlength[4];      /* Runlength encoded flag */
    char author[48];        /* Name of who made it */
    char date[32];          /* Date and time image was made */
    char program[16];       /* Program that created this file */
    char comment[96];       /* other viewing info. for this image */
    unsigned char c_map[256][3]; /* RGB values of the pixmap indices */
} ImageHeader;


/* Note:
 * - All data is in char's in order to maintain easily portability
```

```
*   across machines and some human readibility.
* - Images may be stored as pixmaps or in seperate channels, such as
*   red, green, blue data.
* - An optional alpha channel is seperate and is found after every
*   num_channels of data.
* - Pixmaps, red, green, blue, alpha and other channel data are stored
*   sequentially after the header.
* - If num_channels = 1 or 0, a pixmap is assumed and up to num_colors
*   of colormap in the header are used.
*/

/*** end ImageHeader.h ***/
```

include/ImageP.h

```
/*
* $XConsortium: ImageP.h,v 1.24 89/06/08 18:05:01 swick Exp $
*/
```

```
TORTIOUS ACTION,
ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF
THIS
SOFTWARE.

******************************************************************/

/*
 * ImageP.h - Private definitions for Image widget
 *
 */

#ifndef _XawImageP_h
#define _XawImageP_h

/******************************************************************
 *
 * Image Widget Private Data
 *
 ******************************************************************/

#include "../include/Image.h"
#include <X11/Xaw/SimpleP.h>

/* New fields for the Image widget class record */

typedef struct {int foo;} ImageClassPart;

/* Full class record declaration */
typedef struct _ImageClassRec {
    CoreClassPart      core_class;
    SimpleClassPart    simple_class;
```

```
   ImageClassPart    image_class;
} ImageClassRec;

extern ImageClassRec imageClassRec;

/* New fields for the Image widget record */
typedef struct {
   /* resources */
       Pixmap        pixmap;
       XtCallbackList       callbacks;

   /* private state */
       Dimension     map_width, map_height;
} ImagePart;


/*****************************************************************
 *
 * Full instance record declaration
 *
 *****************************************************************/

typedef struct _ImageRec {
   CorePart   core;
   SimplePart         simple;
   ImagePart  image;
} ImageRec;

#endif /* _XawImageP_h */
```

include/Message.h

```
typedef struct {
    Widget shell, widget; /* shell and text widgets (NULL if not created */
    XawTextBlock info; /* Display text */
    int size, rows, cols; /* Size of buffer (info.ptr) & dimensions of display */
    XawTextEditType edit; /* edit type */
    Boolean own_text; /* text is owned by message? */
} MessageRec, *Message;
```

include/Palette.h

```
#define PalettePath "."
#define     PaletteExt     ".pal"

typedef     struct  _MapRec {
        int     start, finish, m, c;
        struct  _MapRec     *next;
} MapRec, *Map;

typedef     struct  _PaletteRec {
        char    name[STRLEN];
        Map     mappings;
        struct  _PaletteRec     *next;
} PaletteRec, *Palette;
```

include/PullRightMenu.h

```
/*
 * $XConsortium: PullRightMenu.h,v 1.17 89/12/11 15:01:55 kit Exp $
 *
 * Copyright 1989 Massachusetts Institute of Technology
 *
 * Permission to use, copy, modify, distribute, and sell this software and its
 * documentation for any purpose is hereby granted without fee, provided that
 * the above copyright notice appear in all copies and that both that
 * copyright notice and this permission notice appear in supporting
 * documentation, and that the name of M.I.T. not be used in advertising or
 * publicity pertaining to distribution of the software without specific,
 * written prior permission.  M.I.T. makes no representations about the
 * suitability of this software for any purpose.  It is provided "as is"
 * without express or implied warranty.
 *
 * M.I.T. DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL M.I.T.
 * BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES
OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION
 * OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN
 * CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 *
 */
```

```
/*
 * PullRightMenu.h - Public Header file for PullRightMenu widget.
 *
 * This is the public header file for the Athena PullRightMenu widget.
 * It is intended to provide one pane pulldown and popup menus within
 * the framework of the X Toolkit.  As the name implies it is a first and
 * by no means complete implementation of menu code. It does not attempt to
 * fill the needs of all applications, but does allow a resource oriented
 * interface to menus.
 *
 */

#ifndef _PullRightMenu_h
#define _PullRightMenu_h

#include <X11/Shell.h>
#include <X11/Xmu/Converters.h>

/****************************************************************
 *
 * PullRightMenu widget
 *
 ****************************************************************/

/* PullRightMenu Resources:

 Name                Class               RepType     Default Value
 ----                -----               -------     -------------
 background          Background          Pixel       XtDefaultBackground
 backgroundPixmap    BackgroundPixmap    Pixmap      None
 borderColor         BorderColor         Pixel       XtDefaultForeground
 borderPixmap        BorderPixmap        Pixmap      None
```

borderWidth	BorderWidth	Dimension	1
bottomMargin	VerticalMargins	Dimension	VerticalSpace
columnWidth	ColumnWidth	Dimension	Width of widest text
cursor	Cursor	Cursor	None
destroyCallback	Callback	Pointer	NULL
height	Height	Dimension	0
label	Label	String	NULL (No label)
labelClass	LabelClass	Pointer	smeBSBObjectClass
mappedWhenManaged	MappedWhenManaged	Boolean	True
rowHeight	RowHeight	Dimension	Height of Font
sensitive	Sensitive	Boolean	True
topMargin	VerticalMargins	Dimension	VerticalSpace
width	Width	Dimension	0
button	Widget	Widget NULL	
x	Position	Position	0
y	Position	Position	0

```
*/

typedef struct _PullRightMenuClassRec*    PullRightMenuWidgetClass;
typedef struct _PullRightMenuRec*         PullRightMenuWidget;

extern WidgetClass pullRightMenuWidgetClass;

#define XtNcursor "cursor"
#define XtNbottomMargin "bottomMargin"
#define XtNcolumnWidth "columnWidth"
#define XtNlabelClass "labelClass"
#define XtNmenuOnScreen "menuOnScreen"
#define XtNpopupOnEntry "popupOnEntry"
#define XtNrowHeight "rowHeight"
#define XtNtopMargin "topMargin"
```

```
#define XtNbutton    "button"

#define XtCColumnWidth "ColumnWidth"
#define XtCLabelClass "LabelClass"
#define XtCMenuOnScreen "MenuOnScreen"
#define XtCPopupOnEntry "PopupOnEntry"
#define XtCRowHeight "RowHeight"
#define XtCVerticalMargins "VerticalMargins"
#define         XtCWidget    "Widget"

/****************************************************************
 *
 * Public Functions.
 *
 ****************************************************************/

/*      Function Name: XawPullRightMenuAddGlobalActions
 *      Description: adds the global actions to the simple menu widget.
 *      Arguments: app_con - the appcontext.
 *      Returns: none.
 */

void
XawpullRightMenuAddGlobalActions(/* app_con */);
/*
XtAppContext app_con;
*/

#endif /* _PullRightMenu_h */
```

include/SmeBSBpr.h

```
/*
 * $XConsortium: SmeBSB.h,v 1.5 89/12/11 15:20:14 kit Exp $
 *
 * Copyright 1989 Massachusetts Institute of Technology
 *
 * Permission to use, copy, modify, distribute, and sell this software and its
 * documentation for any purpose is hereby granted without fee, provided that
 * the above copyright notice appear in all copies and that both that
 * copyright notice and this permission notice appear in supporting
 * documentation, and that the name of M.I.T. not be used in advertising or
 * publicity pertaining to distribution of the software without specific,
 * written prior permission.  M.I.T. makes no representations about the
 * suitability of this software for any purpose.  It is provided "as is"
 * without express or implied warranty.
 *
 * M.I.T. DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL M.I.T.
 * BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES
OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION
 * OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN
 * CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 */
```

```
/*
 * SmeBSBpr.h - Public Header file for SmeBSB object.
 *
 * This is the public header file for the Athena BSB Sme object.
 * It is intended to be used with the simple menu widget.  This object
 * provides bitmap - string - bitmap style entries.
 *
 */

#ifndef _SmeBSBpr_h
#define _SmeBSBpr_h

#include <X11/Xmu/Converters.h>

#include <X11/Xaw/Sme.h>

/****************************************************************
 *
 * SmeBSBpr object
 *
 ****************************************************************/

/* BSB pull-right Menu Entry Resources:

 Name             Class         RepType        Default Value
 ----             -----         -------        -------------
 callback         Callback      Callback       NULL
 destroyCallback  Callback      Pointer        NULL
 font             Font          XFontStruct *  XtDefaultFont
 foreground       Foreground    Pixel          XtDefaultForeground
 height           Height        Dimension      0
 label            Label         String         Name of entry
```

```
leftBitmap      LeftBitmap         Pixmap      None
leftMargin      HorizontalMargins  Dimension   4
rightBitmap     RightBitmap        Pixmap      None
rightMargin     HorizontalMargins  Dimension   4
sensitive       Sensitive          Boolean     True
vertSpace       VertSpace          int         25
width           Width              Dimension   0
x               Position           Position    0n
y               Position           Position    0
menuName  MenuName  String  "menu"

*/

typedef struct _SmeBSBprClassRec    *SmeBSBprObjectClass;
typedef struct _SmeBSBprRec         *SmeBSBprObject;

extern WidgetClass smeBSBprObjectClass;

#define XtNleftBitmap "leftBitmap"
#define XtNleftMargin "leftMargin"
#define XtNrightBitmap "rightBitmap"
#define XtNrightMargin "rightMargin"
#define XtNvertSpace   "vertSpace"
#define       XtNmenuName       "menuName"

#define XtCLeftBitmap "LeftBitmap"
#define XtCHorizontalMargins "HorizontalMargins"
#define XtCRightBitmap "RightBitmap"
#define XtCVertSpace   "VertSpace"
#define       XtCMenuName       "MenuName"

#endif /* _SmeBSBpr_h */
```

include/SmeBSBprP.h

```
/*
 * $XConsortium: SmeBSBP.h,v 1.6 89/12/11 15:20:15 kit Exp $
 *
 * Copyright 1989 Massachusetts Institute of Technology
 *
 * Permission to use, copy, modify, distribute, and sell this software and its
 * documentation for any purpose is hereby granted without fee, provided that
 * the above copyright notice appear in all copies and that both that
 * copyright notice and this permission notice appear in supporting
 * documentation, and that the name of M.I.T. not be used in advertising or
 * publicity pertaining to distribution of the software without specific,
 * written prior permission.  M.I.T. makes no representations about the
 * suitability of this software for any purpose.  It is provided "as is"
 * without express or implied warranty.
 *
 * M.I.T. DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL M.I.T.
 * BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES
OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION
 * OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN
 * CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 *
 * Author:  Chris D. Peterson, MIT X Consortium
```

```
*/

/*
 * SmeP.h - Private definitions for Sme object
 *
 */

#ifndef _XawSmeBSBP_h
#define _XawSmeBSBP_h

/***********************************************************************
 *
 * Sme Object Private Data
 *
 ***********************************************************************/

#include <X11/Xaw/SmeP.h>
#include "../include/SmeBSBpr.h"

/************************************************************
 *
 * New fields for the Sme Object class record.
 *
 ************************************************************/

typedef struct _SmeBSBprClassPart {
  XtPointer extension;
} SmeBSBprClassPart;

/* Full class record declaration */
typedef struct _SmeBSBprClassRec {
    RectObjClassPart        rect_class;
```

```
    SmeClassPart      sme_class;
    SmeBSBprClassPart  sme_bsb_class;
} SmeBSBprClassRec;

extern SmeBSBprClassRec smeBSBprClassRec;

/* New fields for the Sme Object record */
typedef struct {
    /* resources */
    String label;               /* The entry label. */
    int vert_space;             /* extra vert space to leave, as a percentage
                                   of the font height of the label. */
    Pixmap left_bitmap, right_bitmap; /* bitmaps to show. */
    Dimension left_margin, right_margin; /* left and right margins. */
    Pixel foreground;           /* foreground color. */
    XFontStruct * font;              /* The font to show label in. */
    XtJustify justify;          /* Justification for the label. */
        String menu_name;       /* Popup menu name */

/* private resources. */

    Boolean set_values_area_cleared; /* Remember if we need to unhighlight. */
    GC norm_gc;                      /* noral color gc. */
    GC rev_gc;                       /* reverse color gc. */
    GC norm_gray_gc;                 /* Normal color (grayed out) gc. */
    GC invert_gc;               /* gc for flipping colors. */

    Dimension left_bitmap_width; /* size of each bitmap. */
    Dimension left_bitmap_height;
    Dimension right_bitmap_width;
    Dimension right_bitmap_height;
```

```
} SmeBSBprPart;

/**************************************************************
 *
 * Full instance record declaration
 *
 **************************************************************/

typedef struct _SmeBSBprRec {
  ObjectPart        object;
  RectObjPart       rectangle;
  SmePart           sme;
  SmeBSBprPart      sme_bsb;
} SmeBSBprRec;

/**************************************************************
 *
 * Private declarations.
 *
 **************************************************************/

#endif /* _XawSmeBSBPpr_h */
```

include/xwave.h

```
#include        <X11/Xlib.h>
#include        <X11/Xutil.h>
#include        <X11/Xatom.h>
#include        <X11/Xaw/Cardinals.h>
#include        <X11/StringDefs.h>
#include        <X11/Xmu/Xmu.h>
#include        <X11/Xaw/Command.h>
#include        <X11/Xaw/List.h>
#include        <X11/Xaw/Box.h>
#include        <X11/Xaw/Form.h>
#include        <X11/Xaw/Scrollbar.h>
#include        <X11/Xaw/Viewport.h>
#include        <X11/Xaw/AsciiText.h>
#include        <X11/Xaw/Dialog.h>
#include        <X11/Xaw/MenuButton.h>
#include        <X11/Xaw/SimpleMenu.h>
#include        <X11/Xaw/SmeBSB.h>
#include        <X11/Xaw/Toggle.h>
#include        "SmeBSBpr.h"
#include        "PullRightMenu.h"
#include        <X11/Shell.h>
#include        <X11/cursorfont.h>
#define         STRLEN      100
#define         NAME_LEN  20
#include        "Image.h"
#include        "Message.h"
#include        <dirent.h>
#include        <math.h>
```

```
#include	<stdio.h>
#include	"Palette.h"
#include	"Icon.h"

#define	PLOT_DIR	"graphs"
#define	PLOT_EXT	".plot"
#define	ELLA_IN_DIR	"."
#define	ELLA_IN_EXT	".eli"
#define	ELLA_OUT_DIR	"."
#define	ELLA_OUT_EXT	".elo"
#define	VID_DIR	"videos"
#define	VID_EXT	".vid"
#define	IMAGE_DIR	"images"
#define	BATCH_DIR	"batch"
#define BATCH_EXT	".bat"
#define	KLICS_DIR	"import"
#define	KLICS_EXT	".klics"
#define	KLICS_SA_DIR	"import"
#define	KLICS_SA_EXT	".klicsSA"

typedef enum {
	TRANS_None, TRANS_Wave,
} TransType;

typedef	enum {
	MONO, RGB, YUV,
} VideoFormat;

extern String ChannelName[3][4];

#define	negif(bool,value)	((bool)?-(value):(value))
```

```
typedef        struct  {
        String  name;
        Pixmap          pixmap;
        unsigned int    height, width;
} IconRec, *Icon;

typedef        void    (*Proc)();
typedef        String  *(*ListProc)();
typedef        Boolean         (*BoolProc)();

typedef        struct  {
        String  name;
        WidgetClass widgetClass;
        String  label;
        String  hook;  /* menuName for smeBSBprObjectClass */
} MenuItem;

typedef        struct  {
        String  name, button;
        ListProc        list_proc;
        String  action_name;
        Proc    action_proc;
        caddr_t         action_closure;
} SelectItem, *Selection;

typedef        struct  {
        TransType       type;
        int     space[2];
        Boolean         dirn;
} WaveletTrans;

typedef        union  {
```

```
    TransType     type;
    WaveletTrans          wavelet;
} VideoTrans;

typedef     struct _VideoRec    {
    char    name[STRLEN];                   /* Name of this video name.vid */
    char    path[STRLEN];                         /* Path to frame file(s) */
    char    files[STRLEN];           /* Name of frames files001 if not name */
    VideoFormat type;                /* Type of video (MONO,RGB,YUV) */
    Boolean     disk;  /* Frames reside on disk rather than in memory */
    Boolean     gamma;                           /* Gamma corrected flag */
    Boolean     negative;                  /* Load negative values in data */
    int     rate;                                  /* Frames per second */
    int     start;                          /* Starting frame number */
    int     size[3];  /* Dimensions of video after extraction x, y and z */
    int     UVsample[2];             /* Chrominance sub-sampling x and y */
    int     offset;                  /* Header length */
    int     cols, rows;              /* Dimensions of video as stored */
    int     x_offset, y_offset;  /* Offset of extracted video in stored */
    VideoTrans   trans;                      /* Transform technique used */
    int     precision;              /* Storage precision above 8 bits */
    short   **data[3];                       /* Image data channels */
    struct  _VideoRec    *next;              /* Next video in list */
} VideoRec, *Video;

typedef     struct {
    Video  video;
    char   name[STRLEN];
} VideoCtrlRec, *VideoCtrl;

typedef     struct  _PointRec    {
    int     location[2];
```

```
        int     usage;
        struct _PointRec      *next;
} PointRec, *Point;

typedef         struct  _FrameRec   {
        Widget          shell, image_widget, point_merge_widget;
        Video   video;
        int     zoom, frame, channel, palette;
        Boolean         point_switch, point_merge;
        Point   point;
        Message         msg;
        struct _FrameRec      *next;
} FrameRec, *Frame;

#define         NO_CMAPS  6

typedef         struct _BatchRec        {
        Proc    proc;
        caddr_t         closure, call_data;
        struct _BatchRec        *next;
} BatchRec, *Batch;

typedef         struct  {
        char    home[STRLEN];
        XtAppContext            app_con;
        Widget          toplevel;
        int     no_icons;
        Icon    icons;
        Video   videos;
        Frame   frames;
        Point   points;
        Palette         palettes;
```

```
	int	no_pals;
	String	parse_file;
	String	parse_token;
	FILE	*parse_fp;
	XVisualInfo	*visinfo;
	int	levels, rgb_levels, yuv_levels[3];
	Colormap	cmaps[NO_CMAPS];
	String	batch;
	Batch	batch_list;
	Boolean	debug;
	int	dither[16][16];
} GlobalRec, *Global;

typedef	struct {
	Widget	widgets[3];
	int	max, min, *value;
	String	format;
} NumInputRec, *NumInput;

typedef	struct {
	Widget	widgets[2];
	double	max, min, *value;
	String	format;
} FloatInputRec, *FloatInput;

extern Global	global;

/* InitFrame.c */

extern Video FindVideo();

/* Pop2.c */
```

```
extern void   NA();
extern Widget        FindWidget();
extern void   Destroy();
extern void   Free();

/* Storage.c */

extern void   NewFrame();
extern void   GetFrame();
extern void   SaveFrame();
extern void   FreeFrame();
extern void   SaveHeader();
extern Video  CopyHeader();

/* Message.c */

extern void   TextSize();
extern Message       NewMessage();
extern void   MessageWindow();
extern void   CloseMessage();
extern void   Mprintf();
extern void   Dprintf();
extern void   Eprintf();
extern void   Mflush();

/* Icon3.c */

extern void   FillForm();
extern void   FillMenu();
extern Widget        ShellWidget();
extern Widget        FormatWidget();
extern void   SimpleMenu();
```

```
extern int      TextWidth();
extern Icon     FindIcon();
extern void     NumIncDec();
extern void     FloatIncDec();
extern void     ChangeYN();
extern XFontStruct   *FindFont();
```

DATA COMPRESSION AND DECOMPRESSION

GREGORY KNOWLES AND ADRIAN S. LEWIS

M-2357 US

APPENDIX B-1

```
MAC ADDR_COUNTER_COL = (bool:ck,t_reset:reset,STRING[xsize]bit:block_cnt_length)
->
(t_col,bool):
BEGIN
MAKE  BASE_COUNTER_COL:base_counter_col.
JOIN   (ck,reset,block_cnt_length)       ->base_counter_col.

OUTPUT (base_counter_col[1], CASE base_counter_col[2]
                            OF          count_carry:t
                            ELSE f
                            ESAC)
END.

MAC ADDR_COUNTER_ROW = (bool:ck,t_reset:reset,STRING[ysize]bit:block_cnt_length,bool:col_carry)
                          ->
                          (t_row,bool):
BEGIN
MAKE   BASE_COUNTER_ROW:base_counter_row.
JOIN   (ck,reset,col_carry,block_cnt_length,CASE col_carry       #type conversion#
                                OF t:count_carry
                                ELSE count_rst
                                ESAC)  ->base_counter_row.

OUTPUT (base_counter_row[1],  CASE base_counter_row[2]
                  OF       count_carry:t
                  ELSE f
                  ESAC)

END.

#the string base address calculators#
```

```
MAC NOMULT_MAC_READ = (bool:ck,t_reset:reset,bool:col_end,t_mux4:mux_control,STRING[17]bit:incr,
        STRING[17]bit:oct_add_factor, STRING[19]bit:base_u base_v)
        ->
        STRING[19]bit:
BEGIN
MAKE  ADD_US_ACTEL{19,17}:add,
        MUX_2{STRING[17]bit}:mux.

LET
   next_addr = MUX_4{STRING[19]bit}(add[2..20],ZERO{19}b"0",base_u,base_v,mux_control),
        dff = DFF_NO_LOAD{STRING[19]bit}(ck,reset,next_addr,b"0000000000000000000").


JOIN    (dff,mux,b'1)     ->add,
        (incr,oct_add_factor,CASE col_end
                           OF  t:right
                           ELSE  left
                           ESAC)            ->mux.
OUTPUT       dff
END.

MAC S_SPA =(STRING[19]bit:in)
->
                     (flag,t_sparc_addr):BIOP TRANSFORM_US.
MAC SPA_S =(t_sparc_addr:in)
->
                     (flag,STRING[19]bit):BIOP TRANSFORM_US.


MAC SPARC_ADDR= (bool:ck,t_reset:reset,bool:col_end,t_mux4:mux_control,[2]t_sparc_addr:oct_add_factor,
```

```
        STRING[19]bit:base_u base_v)
                        ->
                                t_sparc_addr:
BEGIN
LET out=NOMULT_MAC_READ(ck,reset,col_end,mux_control,(SPA_S oct_add_factor[1])[2][3..19],
            (SPA_S oct_add_factor[2])[2][3..19],base_u,base_v).
OUTPUT  (S_SPA out)[2]
END.


#-------------------------------------------------------------------------#


#the read and write address generator,input the initial image & block sizes for oct/0 at that channel#
FN ADDR_GEN_NOSCRATCH= (bool:ck,t_reset:reset,t_direction:direction,t_channel:channel,
    STRING[9]bit:x_p_1,STRING[11]bit:x3_p_1,STRING[12]bit:x7_p_1,
        STRING [ysize]bit:octave_row_length,STRING [xsize]bit:octave_col_length,t_reset:octave_reset,
        t_octave:octave,bool:y_done,bool:uv_done, t_load:octave_finished, STRING [19]bit:base_u base_v)

            ->
        ((t_input_mux,t_sparcport,t_dwtport#dwt#),t_load#IDWT data valid#,t_load#read_valid#
                ,t_count_control#row read col read#,(t_col,t_count_control)#addr_col_read#):
#the current octave and when the block finishes the 3 octave transform#
BEGIN

#    ADDR_COUNTER_ROW:addr_row_write,#
#        ADDR_COUNTER_COL:addr_col_write,#
MAKE      ROW_COUNT_CARRY:addr_row_read,
          COL_COUNT:        addr_col_read,
          SPARC_ADDR:write_addr read_addr,
        MEM_CONTROL_NOSCRATCH:mem_control,
```

```
        JKFF:zero_hh_bool read_done_bool.          #write begins #

LET   mem_sel                    = CASE octave
                           OF    oct/0:uno,
                                 oct/1:dos,
                                 oct/2:tres,
                                 oct/3:quatro
                           ESAC,

      sparc_add_1           = MUX_4{t_sparc_addr}(
                            (addr/1),
                            (addr/2),
                            (addr/4),
                            (addr/8),
                            mem_sel),

      sparc_add_2_y        = MUX_4{STRING[12]bit}(
                  (b"000000000001"),
                  (b"000" CONC x_p_1[1..7] CONC b"10" ),
                  (b"0" CONC x3_p_1[1..8] CONC b"100" ),
                  (x7_p_1[1..8] CONC b"1000" ),
                            mem_sel),

     sparc_add_2_uv        = MUX_4{STRING[12]bit}(
                  (b"000000000001"),
                  (b"0000" CONC x_p_1[1..6] CONC b"10"),
                  (b"00" CONC x3_p_1[1..7] CONC b"100"),
                  (b"0" CONC x7_p_1[1..7] CONC b"1000"),
                  mem_sel),

      sparc_add_2           = MUX_2{STRING[12]bit}( sparc_add_2_y, sparc_add_2_uv, CASE channel
```

```
                                                        OF      y:left
                                                        ELSE    right
                                                        ESAC),

sparc_oct_add_factor = (sparc_add_1,(S_SPA( b"0000000" CONC sparc_add_2))[2]),

#signals when write must start  delayed 1 tu for use in zero_hh#

  addr_col_read_flag =CASE addr_col_read[2]#decode to bool#
            OF   count_carry:t
            ELSE f
            ESAC,

  write_latency =  CASE (addr_row_read[1], addr_col_read[1])
                   OF     (row/2,col/(conv2d_latency-1)):t
                      ELSE f
          ESAC,

    read_done = CASE (addr_row_read[2], addr_col_read_flag)        #read input data done#
                   OF     (count_carry,t):t
                   ELSE f
                   ESAC,

    zero_hh = CAST[t_load](NOT zero_hh_bool),

    read_valid= CAST[t_load](NOT read_done_bool),

    start_write_col= DFF_NO_LOAD[t_load](ck,reset,zero_hh,read),         #1 tu after zero_hh#
```

```
read_mux =      CASE  (y_done,uv_done,octave_finished,channel)
                OF      (t,f,write,y)|(f,f,write,u):tres,        #base_u#
                        (f,t,write,u)|(f,f,write,v):quatro,      #base v#
                        (f,bool,write,y):dos                              #base y#
                ELSE    uno
                ESAC,

write_mux =   CASE  zero_hh
        OF    write: uno,
                  read:       CASE channel
                              OF      y:dos,          #base y#
                                      u:tres,         #base_u#
                                      v:quatro        #base v#
                              ESAC
              ESAC.

JOIN
#note that all the counters have to be reset at the end of an octave, ie on octave_finished#
     (ck,octave_reset,octave_col_length)   ->addr_col_read,      #the row&col counts for the read address#
     (ck,octave_reset,octave_row_length,addr_col_read[2])          ->addr_row_read,

        (ck,octave_reset,write_latency,t) ->zero_hh_bool,

        (ck,octave_reset,read_done,t)              ->read_done_bool,

#w&r addresses for sparc mem#
     (ck,reset,PDF1{bool,conv2d_latency-1}(ck,reset,addr_col_read_flag ,f) ,write_mux,sparc_oct_add_factor,base_u,base_v)
                  ->write_addr,
```

```
        (ck,reset,addr_col_read_flag,read_mux,sparc_oct_add_factor,base_u,base_v)     ->read_addr,

        (ck,reset,direction,channel,octave,write_addr,read_addr,zero_hh)->mem_control.

OUTPUT( mem_control,zero_hh, read_valid,addr_row_read[2],addr_col_read)
END.
#the basic 2d convolver for transform, rows first then cols.#
FN CONV_2D = (bool:ck,t_reset:reset, t_input:in, t_direction:direction, [4]t_scratch:pdel,
        t_reset:conv_reset,t_count_control:row_flag,(t_col,t_count_control):addr_col_read)
                ->
            (t_input,t_memport,t_count_control,t_count_control,t_count_control):

#forward direction outputs in row form      #
#       HH HG HH HG .....                   #
#       HG GG HG GG.....                    #
#       HH HG HH HG .....                   #
#       HG GG HG GG.....                    #
#the inverse convolver returns the raster scan format output data#
#the convolver automatically returns a 3 octave transform#

BEGIN
        FN CH_PORT = (([4]t_scratch,t_col),t_col)
        ->
            t_memport:REFORM.

MAKE    CONV_ROW:conv_row,
        CONV_COL:conv_col.

LET
```

```
row_reset =    CASE direction
           OF  forward:conv_reset,
               inverse: PDF1{t_reset,1}(ck,no_rst,conv_reset,rst)     #pipeline delays in col_conv#
           ESAC,
    col_reset =      CASE direction
                     OF   forward:PDF1{t_reset,3}(ck,no_rst,conv_reset,rst),
                          inverse: conv_reset                        #pipeline delays in row_conv#
                     ESAC,

col_flag = DFM{t_count_control}(ck,addr_col_read[2],PDF1{t_count_control,1}(ck,reset,addr_col_read[2],
                         count_0), CAST{bool}direction),

row_control = DFM{t_count_control}(ck,PDF1{t_count_control,3}(ck,reset,row_flag,count_0),
                         row_flag, CAST{bool}direction),

direction_sel =CASE direction          #mux control for the in/out data mux's#
         OF    forward:left,
               inverse:right
         ESAC,

    col_count = MUX_2{(t_col,t_count_control)}(
                     PDF1{(t_col,t_count_control),3}(ck,reset,addr_col_read,(col/0,count_rst)),
                     addr_col_read,
                     direction_sel),

#pipeline delays for the convolver values and input value#
    del_conv_col=DFF_NO_LOAD{t_input}(ck,reset,conv_col[1],input/0),

    del_conv_row=DFF_NO_LOAD{t_input}(ck,reset,conv_row,input/0),

    del_in = DFF_NO_LOAD{t_input}(ck,reset,in,input/0).
```

```
JOIN

(ck,row_reset,direction,MUX_2{t_input}(del_in,del_conv_col,direction_sel), col_flag)    ->conv_row,

(ck,col_reset,direction,MUX_2{t_input}(del_conv_row,del_in,direction_sel), pdel,row_control,col_count) ->conv_col.


OUTPUT  (MUX_2{t_input}(del_conv_col,del_conv_row, direction_sel) ,CH_PORT(conv_col[2],col_count[1]),row_control,col_count[2],col_flag)
END.

# 1d col convolver, with control #

FN CONV_COL = (bool:ck,t_reset:reset, t_direction:direction, t_input:in,
[4]t_scratch:pdel,t_count_control:row_flag,
(t_col,t_count_control):col_count)
                                    ->
                    (t_input,([4]t_scratch,t_col)):

#input is data in and, pdel, out from line-delay memories#
# out is (G,H), and line delay out port.  The row counter is started 1 cycle later to allow for#
#pipeline delay between MULTIPLIER and this unit #

BEGIN
# a %2 line by line resetable counter for the state machines, out->one on rst#
#carry active on last element of row#
    MAC COUNT_2 = (bool:ck,t_reset:reset,t_count_control:carry)
                                    ->
                                          t_count_2:
    BEGIN
```

```
    MAKE DFF_NO_LOAD{t_count_2}:countdel.
  LET    countout= CASE (countdel,carry)
              OF     (one,count_carry):two,
                     (two,count_carry):one
              ELSE countdel
              ESAC.
  JOIN   (ck,reset,countout,one)    ->countdel.
  OUTPUT countdel
  END.


#the code for the convolver#
MAKE   MULT_ADD:mult_add,
       [4]DF1{t_scratch}:pdel_in,
       [4]DF1{t_ssratch}:pdel_out,
       COUNT_2:count.

# now the state machines to control the convolver#
#First the and gates#

LET
       reset_row=DF1{t_reset}(ck,reset),          #starts row counter 1 cycle after frame start#
#we want the row counter to be 1 cycle behind the col counter for the delay for the#
#pipelined line delay memory#

col_carry =DFF_NO_LOAD{t_count_control}(ck,reset,col_count[2],count_rst),

#these need to be synchronised to keep the row counter aligned with the data stream#
#also the delay on col_count deglitches the col carryout#

       row_control=row_flag,              #signal for row=0,1,2,3, last row, etc#
```

```
andsel=(CASE direction
        OF      forward: CASE count
                                OF      one:pass,
                                        two:zero
                          ESAC,
                inverse: CASE count
                          OF      one:zero,
                                  two:pass
                          ESAC
        ESAC,

         CASE row_control
        OF count_0:zero
        ELSE pass
        ESAC,

        CASE direction
        OF      forward: CASE row_control
                          OF count_0:zero
                          ELSE pass
                          ESAC,
                inverse: pass
        ESAC),
#now the add/sub control for the convolver adders#
        addsel= CASE count
        OF one:(add,add,add,sub),
          two:(add,sub,add,add)
        ESAC,
```

```
#now the mux control#
	centermuxsel=		CASE direction
				OF	forward: CASE count
						OF one:(left,right),
						  two:(right,left)
					ESAC,
				inverse:CASE count
					OF	one:(right,left),
						two:(left,right)
					ESAC
				ESAC,
#the perfect reconstruction output#
#the addmuxsel signal#
	muxandsel =		CASE direction
				OF	forward:(andsel[2],pass,andsel[2]),
					inverse:(pass,andsel[2], CASE row_control
								OF	count_1:zero
								ELSE	pass
								ESAC)
				ESAC,
	muxsel=		CASE direction
		OF	forward:(uno,

				CASE row_control
				OF count_0:dos,
				   count_carry:tres
				ELSE uno
				ESAC,

				CASE row_control
				OF count_0:tres,
```

```
                         count_carry:quatro
                       ELSE dos
                       ESAC),

                 inverse:( CASE row_control
                       OF count_0:dos,
                         count_1:quatro,
                         count_carry:dos,
                         count_lm1:tres
                       ELSE dos
                       ESAC,

                       CASE row_control
                       OF count_0:tres,
                         count_carry:dos
                       ELSE uno
                       ESAC,

                       uno)
              ESAC.
LET
#ACTEL#
      wr_addr        =DF1{t_col}(ck,DF1{t_col}(ck,col_count[1]) ),
                                        #need 2 delays between wr and rd addr#
      rd_addr=col_count[1].             #address for line delay memory#


#join the control signals to the mult_add block#
JOIN  (ck,reset_row,col_carry) ->count,

      (ck,reset,in,andsel,centermuxsel,muxsel,muxandsel,addsel,direction,pdel_out)    ->mult_add.
```

```
FOR INT k=1..4 JOIN
        (ck,mult_add[k]) ->pdel_in[k],      #delay to catch the write address#
        (ck,pdel[k])        ->pdel_out[k].              #read delay to match MULT delay#

#ACTEL HACK#
LET gh_select =    CASE (direction,DF1{t_count_2}(ck,count) )
            OF      (inverse,one)|(forward,two):right,
                  (inverse,two)|(forward,one):left
            ESAC,

   gh_out = MUX_2{t_scratch}(pdel_in[4],DF1{t_scratch}(ck,pdel_out[1]),gh_select),
   shift_const= CASE direction
        OF    inverse:        CASE DF1{t_count_control}(ck,row_control)
                          OF     (count_1 | count_2):shift3
                          ELSE  shift4
                          ESAC,
            forward:       shift5
            ESAC.
OUTPUT  (ROUND_BITS(gh_out,shift_const),(pdel_in,wr_addr#rd_addr#))          #LOCAMII#
END.
#the 1d convolver, with control and coeff extend#


FN CONV_ROW =(bool:ck,t_reset:reset,t_direction:direction,t_input:in, t_count_control:col_flag)
                                          ->
                                                  t_input:
# out is (G,H).  The row counter is started 1 cycle later to allow for#
#pipeline delay between MULTIPLIER and this unit #
#the strings give the col & row lengths for this octave#
```

```
BEGIN
        # a %2 line by line resetable counter for the state machines, out->one on rst#

        MAC COUNT_2 = (bool:ck,t_reset:reset)
                                                ->
                                                        t_count_2:
        BEGIN
                MAKE DFF_NO_LOAD(t_count_2):countdel.
        LET     countout= CASE (countdel)
                         OF     (one):two,
                                (two):one
                         ESAC.
        JOIN    (ck,reset,countout,one)  ->countdel.
        OUTPUT          countdel
        END.

#the code for the convolver#
MAKE    MULT_ADD:mult_add,
        [4]DF1{t_scratch}:pdel,
        COUNT_2:count.

# now the state machines to control the convolver#
#First the and gates#

LET

        reset_col=DF1{t_reset}(ck,reset),        #starts row counter 1 cycle after frame start#
                                        #makes up for the pipeline delay in MULT#
#IIILATENCY DEOENDENTII#
        col_control=col_flag,           #flag when col_count=0,1,2,col_length,etc#
```

```
andsel=(CASE direction
        OF      forward: CASE count
                                OF      one:pass,
                                        two:zero
                        ESAC,
                inverse: CASE count
                        OF      one:zero,
                                two:pass
                        ESAC
        ESAC,

         CASE col_control
        OF count_0:zero
        ELSE pass
        ESAC,

        CASE direction
        OF      forward: CASE col_control
                        OF count_0:zero
                        ELSE pass
                        ESAC,
                inverse: pass
        ESAC),
#now the add/sub control for the convolver adders#
    addsel= CASE count
        OF one:(add,add,add,sub),
          two: (add,sub,add,add)
        ESAC,

#now the mux control#
```

```
centermuxsel=           CASE direction
                        OF      forward: CASE count
                                              OF one:(left,right),
                                                two:(right,left)
                                         ESAC,
                                inverse:CASE count
                                         OF     one:(right,left),
                                                two:(left,right)
                                         ESAC
                        ESAC,
#the addmuxsel signal#
    muxandsel =         CASE direction
                        OF      forward:(andsel[2],pass,andsel[2]),
                                inverse:(pass,andsel[2], CASE col_control
                                                         OF     count_1:zero
                                                         ELSE   pass
                                                         ESAC)
                        ESAC,
    muxsel=         CASE direction
            OF      forward:(uno,

                        CASE col_control
                        OF count_0:dos,
                          count_carry:tres
                        ELSE uno
                        ESAC,

                        CASE col_control
                        OF count_0:tres,
                          count_carry:quatro
                        ELSE dos
```

```
                ESAC),

            inverse:( CASE col_control
                OF  count_0:dos,
                    count_1:quatro,
                    count_lm1:tres
                ELSE dos
                ESAC,

                CASE col_control
                OF  count_0:tres,
                    count_carry:dos
                ELSE uno
                ESAC,

                uno)
        ESAC.


#join the control signals to the mult_add block#
JOIN    (ck,reset_col)    ->count,
#set up the col counters #
        (ck,reset,in,andsel,centermuxsel,muxsel,muxandsel,addsel,direction,pdel)->mult_add.

FOR INT j=1..4 JOIN
            (ck,mult_add[j]) ->pdel[j].          #pipeline delay for mult-add unit#

#ACTEL HACK#
LET     gh_select=CASE direction
            OF    inverse:     CASE count
                          OF    one: left,
```

```
                    two: right
                ESAC,
        forward:    CASE count
                OF   one:right,
                    two:left
                ESAC
    ESAC,

    gh_out = MUX_2{t_scratch}(pdel[4],DF1{t_scratch}(ck, pdel[1]),gh_select),

  rb_select= CASE direction
    OF    inverse:CASE col_control
                      OF     (count_2 | count_3):shift3
                      ELSE  shift4
                      ESAC,
          forward:    shift5
          ESAC.
OUTPUT ROUND_BITS(gh_out,rb_select)
END.


#some string macros#
MAC EQ_US = (STRING[INT n]bit: a b)
                      ->
                          bool: BIOP EQ_US.

#ACTEL 8 bit comparitor macro#
FN ICMP8 = (STRING[8]bit: a b)
                      ->
                          bool: EQ_US{8}(a,b).
```

```
#..............................................................#
#A set of boolean ,ie gate level counters                   #
#..............................................................#

#..............................................................#
#The basic toggle flip-flop plus and gate for a synchronous counter  #
#input t is the toggle ,outputs are q and tc (toggle for next counter#
#stage                                                       #
#..............................................................#

MAC  BASIC_COUNT = (bool:ck ,t_reset:reset,bool: tog)
                                                    ->
                                                    [2]bool:

BEGIN
            MAKE DFF_NO_LOAD{bool}:dlat,
                  XOR    :xor,
                  AND    :and.

            JOIN  (ck,reset,xor,f)->dlat,
                  (dlat,tog)    ->and,
                  (tog,dlat)    ->xor.
            OUTPUT (dlat,and)
END.

#..............................................................#
# The n-bit macro counter generator, en is the enable, the outputs  #
#are msb(bit 1).....lsb,carry.This is the same order as ELLA strings are stored#
#..............................................................#

MAC COUNT_SYNC{INT n} = (bool:ck,t_reset: reset,bool: en )
```

```
                                    ->
                                                ([n]bool,bool):
(LET out = BASIC_COUNT(ck,reset,en) .

        OUTPUT        IF n=1
                THEN  ([1]out[1],out[2])
                ELSE ( LET outn = COUNT_SYNC{n-1}(ck,reset,out[2]) .
                        OUTPUT (outn[1] CONC out[1],outn[2])
                )
                FI
).

#a mod 2^xsize counter#
MAC MOD2_COUNTER_COL = (bool:ck,t_reset:reset)
                                    ->
                                            (t_col):
BEGIN
        MAC S_TO_C  = (STRING[xsize]bit:in)
                                    ->
                                            (flag,t_col):BIOP TRANSFORM_US.
        MAKE COUNT_SYNC{xsize}:count,
           BOOL_STRING{xsize}:b_s.

JOIN    (ck,reset,t)        ->count,         #count always enabled#
        count[1]->b_s.
OUTPUT (S_TO_C b_s)[2]
END.


#a mod 2^ysize counter#
MAC MOD2_COUNTER_ROW = (bool:ck,t_reset:reset,bool:en)
```

```
                                                        ->
                                                                (t_row):
BEGIN
    MAC S_TO_R  = (STRING[ysize]bit:in)
                                                  ->
                                                        (flag,t_row):BIOP TRANSFORM_US.
    MAKE COUNT_SYNC{ysize}:count,
       BOOL_STRING{ysize}:b_s.

JOIN   (ck,reset,en)   ->count,
    count[1]       ->b_s.
OUTPUT (S_TO_R b_s)[2]
END.

#the basic mod col_length counter, to be synthesised#
MAC BASE_COUNTER_COL = (bool:ck,t_reset:reset,STRING[xsize]bit:octave_cnt_length)
                                                                          ->
                                                                          (t_col,t_count_control):
BEGIN

       MAC C_TO_S = (t_col: in)
                                           ->
                                                  (flag,STRING[xsize]bit): BIOP TRANSFORM_US.
    MAC FINAL_COUNT = (t_col:in,STRING[xsize]bit:octave_cnt_length)
                                           ->
                                     t_count_control:
    BEGIN
    LET    in_us =  (C_TO_S in)[2],
        lsb=in_us[xsize].
    #OUTPUT CASE EQ_US(in_us[1..xsize-1],octave_cnt_length[1..xsize-1])  the msb's are the same#
#ACTEL#
```

```
  OUTPUT CASE ICMP8(in_us[1..xsize-1],octave_cnt_length[1..xsize-1])  #the msb's are the same#
      OF t: CASE lsb              #so check the lsb#
          OF b'1:count_carry,   #count odd, so  must be length#
            b'0:count_lm1       #count is even so must be length-1#
          ESAC
      ELSE count_rst
      ESAC
    END.
MAKE  MOD2_COUNTER_COL:mod2_count,
      FINAL_COUNT:final_count.

JOIN  (mod2_count,octave_cnt_length)          ->final_count,
      (ck,CASE reset                                    #system reset or delayed carryout reset#
        OF    rst: rst
        ELSECASE DFF_NO_LOAD{t_count_control}(ck,reset,final_count,count_0)  #latch to avoid glitches#
                    OF      count_carry:rst
                    ELSE no_rst
                    ESAC
        ESAC)                                       ->mod2_count.
OUTPUT (mod2_count,final_count)
END.


FN COL_COUNT_ST = (bool:ck,t_reset:reset,STRING[xsize]bit:octave_cnt_length)
                                                  ->
                                          (t_col,t_count_control):
                                    #count value , and flag for count=0,1,2,col_length-1, col_length#
BEGIN
      MAKE BASE_COUNTER_COL:base_col.

LET   count_control = CASE reset
```

```
                OF  rst:count_0
                    ELSE CASE base_col[1]
                    OF      col/0:count_0,
                            col/1:count_1,
                            col/2:count_2,
                            col/3:count_3

                            ELSE   base_col[2]
                            ESAC
                ESAC.
  JOIN (ck, reset,octave_cnt_length)      ->base_col.
  OUTPUT      (base_col[1],count_control)
END.


#the basic mod row_length counter, to be synthesised#
MAC BASE_COUNTER_ROW = (bool:ck,t_reset:reset,bool:en,STRING[ysize]bit:octave_cnt_length,t_count_control:col_carry)
                                        ->
                                        (t_row,t_count_control):
BEGIN

    MAC R_TO_S = (t_row: in)
                                ->
                                        (flag,STRING[ysize]bit): BIOP TRANSFORM_US.
    MAC FINAL_COUNT = (t_row:in,STRING[ysize]bit:octave_cnt_length)
                                            ->
                                                t_count_control:
    BEGIN
    LET    in_us =  (R_TO_S in)[2],
        lsb=in_us[ysize].
    #OUTPUT CASE EQ_US(in_us[1..ysize-1],octave_cnt_length[1..ysize-1])  the msb's are the same#
```

```
#ACTEL#
    OUTPUT CASE ICMP8(in_us[1..ysize-1],octave_cnt_length[1..ysize-1])  #the msb's are the same#
        OF 1: CASE lsb              #so check the lsb#
            OF b'1:count_carry,     #count odd, so  must be length#
              b'0:count_lm1         #count is even so must be length-1#
            ESAC
        ELSE count_rst
        ESAC
    END.
MAKE   MOD2_COUNTER_ROW:mod2_count,
    FINAL_COUNT:final_count.

#need to delay the reset at end of count signal till end of final row#
#WAS DFF WITH reset#
LET  count_reset =DF1{t_reset}(ck,CASE(final_count,col_carry)   #last row/last col#
                                    OF (count_carry,count_carry):rst #latch to avoid glitches#
                                    ELSE no_rst
                                    ESAC).

JOIN   (mod2_count,octave_cnt_length)  ->final_count,
    (ck,CASE reset                      #system reset or delayed carryout reset#
      OF   rst: rst
      ELSE     count_reset
      ESAC,en)                ->mod2_count.
OUTPUT (mod2_count,final_count)
END.


FN ROW_COUNT_CARRY_ST = (bool:ck,t_reset:reset,STRING[ysize]bit:octave_cnt_length,t_count_control:col_carry)
                                         ->
                                    (t_row,t_count_control):
```

```
BEGIN
        MAKE BASE_COUNTER_ROW:base_row.
LET   count_control = CASE reset
              OF rst:count_0
                  ELSE CASE base_row[1]
                  OF    row/0:count_0,
                      row/1:count_1,
                      row/2:count_2,
                      row/3:count_3

                      ELSE   base_row[2]
                      ESAC
              ESAC.

  JOIN (ck,reset,CASE col_carry
                  OF count_carry:t
                  ELSE f
                  ESAC,octave_cnt_length,col_carry)     ->base_row.
  OUTPUT    (base_row[1],count_control)
 END.

#the discrete wavelet transform chip/ multi-octave/2d transform with edge compensation#
#when ext & csl are both low latch the setup params from the nubus(active low), as follows#
#adl[1..4]        select function#
#        0000     load max_octaves, luminance/colour, forward/inversebar#
#        0001     load yimage#
#        0010     load ximage#
#jump table values#
#        0011     load ximage+1#
#        0100     load 3ximage+3#
#        0101     load 7ximage+7#
```

```
#       0110    load base u addr#
#       0111    load base v addr#

#adl[21..22]    max_octaves#
#adl[23]luminance/crominancebar active low, 1 is luminance,0 is colour#
#adl[24]forward/inversebar active low, 1 is forward, 0 is inverse#

#adl[5..24]     data (bit 24 lsb)#

FN ST_OCT = (STRING[2]bit:st)
->
                        (flag,t_octave): BIOP TRANSFORM_US.

FN OCT_ST = (t_octave:st)
->
                        (flag,STRING[2]bit):BIOP TRANSFORM_US.

FN DWT = (bool:ck_in,t_reset:reset_in, t_input:in_in,bool:extwritel_in csl_in, STRING[24]bit:adl,
        t_input:sparc_mem_in, [4]t_scratch:pdel_in)


        ->
                (t_input#out IDWT data#,[3]t_load#valid out IDWT data,y,u,v#,
        [3]t_load#valid in DWT data y,u,v#,
        t_sparcport#sparc_data_addr, etc#,
        t_memport#pdel_data_out#):

BEGIN
MAKE CONV_2D:conv_2d,
        ADDR_GEN_NOSCRATCH:addr_gen,
#active low clock &enable latches#
```

```
        [2]DLE1D:max_octave_st,
        DLE1D:channel_factor_st,
        DLE1D:dir,
        [9]DLE1D:col_length_s,
        [9]DLE1D:row_length_s,
        [9]DLE1D:x_p_1,
        [11]DLE1D:x3_p_1,
        [12]DLE1D:x7_p_1,
        [19]DLE1D:base_u,
        [19]DLE1D:base_v,
#active low 3X8 decoder#
        DEC3X8A                 :decodel,
#the octave control#
        DFF_INIT{t_octave}: octave ,
        DFF_INIT{t_channel}: channel ,
        JKFF:row_carry_ff,
#pads#
        INBUF{STRING[24]bit}:adl_out,
        CLKBUF:ck,
        INBUF{bool}:extwritel csl,
        INBUF{t_reset}:reset,
        INBUF{t_input}:in sparc_mem,
        INBUF{[4]t_scratch}:pdel,
        OBHS{t_input}:out1,
        OBHS{[3]t_load}:out2 out3,
        OBHS{t_sparcport}:out4,
        OBHS{t_memport}:out5.
#must delay the write control to match the data output of conv_2d, ie by conv2d_latency#
LET
#set up the control params#
```

```
    max_oct = (ST_OCT BOOL_STRING{2}max_octave_st)[2],

    channel_factor= CAST{t_channel_factor}channel_factor_st,

    col_length =  BOOL_STRING{9} col_length_s,

    row_length =  BOOL_STRING{9} row_length_s,

  direction =CASE dir
     OF     f:forward,
          t:inverse
     ESAC,

#set up the octave params#
      convcol_row= conv_2d[3],
   convcol_col=conv_2d[4],
   convrow_col=conv_2d[5],
#signals that conv_col, for forward, or conv_row, for inverse, has finished that octave#
#and selects the next octave value and the sub-image sizes#

   octave_finished =CASE direction
           OF     forward:CASE (row_carry_ff,convcol_row,convcol_col)
                    OF  (t,count_2,count_2):write  #row then col, gives write latency#
                    ELSE read
                    ESAC,
                inverse:CASE (row_carry_ff,convcol_row,convrow_col)
                    OF  (t,count_2,count_3):write   #extra row as col then row#
                    ELSE read
                    ESAC
            ESAC,
 #max octaves for u|v#
```

```
max_oct_1       = CASE max_oct
                 OF     oct/1:oct/0,
                        oct/2:oct/1,
                        oct/3:oct/2
                ESAC,

y_done =CASE (channel,(OCT_ST octave)[2] EQ_US CASE direction
                                        OF      forward:CAST{STRING [2]bit}max_octave_st,
                                                inverse:b"00"
                                        ESAC)
        OF      (y,t):t
        ELSE f
        ESAC,

uv_done = CASE (channel,(OCT_ST octave)[2] EQ_US CASE direction
                        OF     forward:(OCT_ST max_oct_1)[2],
                           inverse:b"00"
                        ESAC)
     OF     (u|v,t):t
     ELSE f
     ESAC,


next=       (SEQ
                VAR new_oct:=octave,
                  new_channel:=channel;
                CASE direction
        OF   forward:(CASE octave
                OF      oct/0:new_oct:=oct/1,
                        oct/1:new_oct:=oct/2,
                   oct/2:new_oct:=oct/3
```

```
ESAC;

                    CASE (y_done,uv_done)
                    OF      (t,bool)|(bool,t):new_oct:=oct/0
                    ELSE
                    ESAC
                    ),

inverse:(CASE octave
        OF    oct/3:new_oct:=oct/2,
              oct/2:new_oct:=oct/1,
              oct/1:new_oct:=oct/0
        ESAC;
       CASE channel
       OF     y: CASE octave
                OF  oct/0:CASE channel_factor #watch for colour#
                                          OF        luminance:new_oct:=max_oct
                                          ELSE               new_oct:=max_oct_1
                                          ESAC
                ELSE
              ESAC,
              u:CASE octave
                OF  oct/0:new_oct:=max_oct_1
                ELSE
                ESAC,
              v:CASE octave
                OF  oct/0:new_oct:=max_oct        #move to y#
                ELSE
                ESAC
       ESAC)
```

```
ESAC;
          CASE  channel_factor
          OF      luminance:new_channel:=y,
                  color: (CASE (channel,y_done)
          OF  (y,t):new_channel:=u
          ELSE
          ESAC;
          CASE (channel,uv_done)
          OF  (u,t):new_channel:=v,
             (v,t):new_channel:=y
          ELSE
          ESAC)
       ELSE
       ESAC;
   OUTPUT (new_oct,new_channel)
   ).


octave_sel = CASE (octave,channel)    #the block size divides by 2 every octave#
         OF    (oct/0,y):uno,      #the u|v image starts 1/4 size#
               (oct/1,y)|(oct/0,u|v):dos,
               (oct/2,y)|(oct/1,u|v):tres,
               (oct/3,y)|(oct/2,u|v):quatro
                ESAC,
octave_row_length =MUX_4{STRING [ysize]bit}(row_length,b"0" CONC row_length[1..ysize-1],
                    b"00" CONC row_length[1..ysize-2],
                                b"000" CONC row_length[1..ysize-3],octave_sel),
octave_col_length = MUX_4{STRING [xsize]bit}(col_length,b"0" CONC col_length[1..xsize-1],
                    b"00" CONC col_length[1..xsize-2],
                                b"000" CONC col_length[1..xsize-3],octave_sel),
```

```
#load next octave, either on system reset, or write finished#
    load_octave=        CASE reset
                    OF  rst:write
                    ELSE    octave_finished
                     ESAC,
#reset the convolvers at the end of an octave, ready for the next octave#
#latch pulse to clean it, note 2 reset pulses at frame start#
#cant glitch as reset&octave_finished dont change at similar times#
    conv_reset =  CASE reset
              OF  rst:rst
              ELSE   CASE  DFF_NO_LOAD{t_load}(ck,reset, octave_finished,read)
                  OF     write:rst
                  ELSE   no_rst
                  ESAC
              ESAC,


#latch control data off nubus, latch control is active low#
      gl =      CASE (extwritel,csl)
                OF       (f,f):f
                ELSE   t
                ESAC,


      sparc_w=addr_gen[1][2][1],       #write addresses#

      input_mux=addr_gen[1][1],  #input_mux#

      sparc_r=addr_gen[1][2][2],       #read addresses#

      sparc_rw = addr_gen[1][2][3],
```

```
inverse_out =   CASE (direction,octave)
                OF      (inverse,oct/0):CASE (channel,addr_gen[2])
                                OF      (y,write):(write,read,read),
                                        (u,write):(read,write,read),
                                        (v,write):(read,read,write)
                                ELSE (read,read,read)
                                ESAC,
                        (forward,oct/0):(read,read,read)
                ELSE (read,read,read)
                ESAC,
forward_in =    CASE direction
                OF      forward:CASE (channel,octave,addr_gen[3])
                                OF      (y,oct/0,read):(read,write,write),
                                        (u,oct/0,read):(write,read,write),
                                        (v,oct/0,read):(write,write,read)
                                        ELSE (write,write,write)
                                        ESAC,
                        inverse:(write,write,write)
                ESAC.
JOIN
#in pads#
        ck_in           ->ck,
        reset_in->reset,
        extwritel_in    ->extwritel,
        csl_in          ->csl,
        adl             ->adl_out,
        in_in           ->in,
        sparc_mem_in    ->sparc_mem,
        pdel_in         ->pdel,
#out pads#
```

```
        conv_2d[1]       ->out1,
        inverse_out      ->out2,
        forward_in       ->out3,
        addr_gen[1][2]  ->out4,
     conv_2d[2] ->out5,

#the control section#
        (CAST{bool}adl[4],CAST{bool}adl[3],CAST{bool}adl[2])    ->decodel,        #active low outs#

        (gl,decodel[1],BIT_BOOLadl_out[21])     ->max_octave_st[1],
        (gl,decodel[1],BIT_BOOLadl_out[22])     ->max_octave_st[2],

        (gl,decodel[1],BIT_BOOLadl_out[23])     ->channel_factor_st,
        (gl,decodel[1],BIT_BOOLadl_out[24])     ->dir.
FOR INT j=1..9 JOIN
        (gl,decodel[2],BIT_BOOLadl_out[15+j])    ->col_length_s[j],
        (gl,decodel[3],BIT_BOOLadl_out[15+j])      ->row_length_s[j],
        (gl,decodel[4],BIT_BOOLadl_out[15+j])      ->x_p_1[j].
FOR INT j=1..11 JOIN
        (gl,decodel[5],BIT_BOOLadl_out[13+j])      ->x3_p_1[j].
FOR INT j=1..12 JOIN
        (gl,decodel[6],BIT_BOOLadl_out[12+j])      ->x7_p_1[j].
FOR INT j=1..19 JOIN
        (gl,decodel[7],BIT_BOOLadl_out[5+j])      ->base_u[j].
        (gl,decodel[8],BIT_BOOLadl_out[5+j])      ->base_v[j].

#sets a flag when row counter moves onto next frame#
JOIN
   (ck,conv_reset,CASE convcol_row
         OF   count_carry:t
         ELSE f
```

```
        ESAC,t)                    ->row_carry_ff,

#load the new octave,after the current octave has finished writing#
# on initial reset must load with starting octave value which depends on direction and channel#
    (ck,no_rst,load_octave, CASE reset
                OF  no_rst:next[1]
                ELSE   CASE (direction,channel)  #initial octave#
                    OF  (forward,t_channel):oct/0,
                      (inverse,y):max_oct,
                      (inverse,u|v):max_oct_1
                   ESAC
                ESAC,oct/0)             ->octave,  #next octave#
      (ck,no_rst,load_octave, CASE reset
              OF  no_rst:next[2]
                ELSE y
                ESAC,y)            ->channel,  #next channel#

      (ck,reset,MUX_2{t_input}(in,sparc_mem,CASE input_mux  #input_mux#
                                      OF  dwt_in:left,
                                          sparc_in:right
                                        ESAC)
      ,direction,pdel, conv_reset,addr_gen[4],addr_gen[5])      ->conv_2d,

(ck,reset,direction,channel,BOOL_STRING{9}x_p_1,BOOL_STRING{11}x3_p_1, BOOL_STRING{12}x7_p_1,octave_row_length,
octave_col_length,conv_reset,octave,y_done,uv_done,octave_finished,BOOL_STRING{19}base_u, BOOL_STRING{19}base_v)    ->addr_gen.

OUTPUT        (out1 ,out2,out3,out4,out5)
END.
FN DWT_TEST = (bool:ck_in,t_reset:reset_in, t_input:in_in,bool:extwritel_in csl_in,t_sparc_addr:reg_sel value)
```

```
            ->
                                  (t_input,[3]t_load,[3]t_load):
BEGIN
   FN SPARC_MEM = (t_input:in,t_sparc_addr:wr_addr,t_sparc_addr:rd_addr,t_load:rw_sparc#,t_cs:cs#)
     ->
          t_input:
         RAM(input/0).

     MAKE  DWT:dwt,
           SPARC_MEM:sparc_mem,
           LINE_DELAY{t_scratch}:line_delay.

LET    data_out=dwt[1],
       sparc_port=dwt[4],
       line_delay_port = dwt[5].
JOIN
       (ck_in,reset_in,in_in,extwritel_in, csl_in,(SPA_S reg_sel)[2][16..19]CONC b"1" CONC(NOT_B (SPA_S value)[2]), sparc_mem,line_delay)
                                   ->dwt,
       (data_out,sparc_port[1],sparc_port[2],sparc_port[3]#,sparc_port[4]#)     ->sparc_mem,

       (line_delay_port[1],line_delay_port[2],line_delay_port[3],write)     ->line_delay.

OUTPUT      dwt[1..3]
END.

# some basic macros for the convolver, assume these will#
#be synthesised into leaf cells#
#the actel MX4 mux cell#
FN NOT = (bool:in)
->
bool:CASE in OF t:f,f:t ESAC.
```

```
MAC  MX_4(TYPE ty)=(ty:in1 in2 in3 in4, [2]bool:sel)
->
                              ty:
CASE   sel
OF      (f,f):in1,
        (f,t):in2,
        (t,f):in3,
        (t,t):in4
ESAC.
#the actel GMX4 mux cell#
MAC  GMX4(TYPE ty)=(ty:in1 in2 in3 in4, [2]bool:sel)
        ->
              ty:
CASE   sel
OF     (f,f):in1,
     (f,t):in2,
     (t,f):in3,
     (t,t):in4
ESAC.

MAC  MXT(TYPE ty)=(ty:a b c d, bool:soa sob s1)
        ->
        ty:
CASE   s1
OF     f: CASE soa
          OF    t:b
          ELSE  a
          ESAC,
        t: CASE sob
```

```
        OF  l:d
        ELSE  c
        ESAC
ESAC.
MAC ENCODE4_2 = (t_mux4:in)
          ->
          [2]bool:
CASE in
OF        uno:(f,f),
          dos:(f,t),
          tres:(t,f),
          quatro:(t,t)
ESAC.

MAC ENCODE3_2 = (t_mux3:in)
          ->
          [2]bool:
CASE in
OF      l:(f,f),
      c:(f,t),
      r:(t,f)
ESAC.

FN DEC3X8A = (bool:a b c)
->
[8]bool:
CASE (a,b,c)
OF        (f,f,f):(f,t,t,t,t,t,t,t),
          (t,f,f):(t,f,t,t,t,t,t,t),
          (f,t,f):(t,t,f,t,t,t,t,t),
          (t,t,f):(t,t,t,f,t,t,t,t),
```

```
        (f,f,t):(t,t,t,t,f,t,t,t),
        (t,f,t):(t,t,t,t,t,f,t,t),
        (f,t,t):(t,t,t,t,t,t,f,t),
        (t,t,t):(t,t,t,t,t,t,t,f)
ESAC.

MAC MUX_2(TYPE t)=(t:in1 in2, t_mux:sel)
->
t:
CASE sel
OF      left:in1,
        right:in2
ESAC.

MAC MUX_3(TYPE t)=(t:in1 in2 in3 , t_mux3:sel)
->
t:
MX_4{t}(in1,in2,in3,in1,ENCODE3_2 sel).

COM
MAC MUX_4(TYPE t)=(t:in1 in2 in3 in4, t_mux4:sel)
->
t:
CASE sel
OF      uno:in1,
        dos:in2,
        tres:in3,
        quatro:in4
ESAC.
MOC
```

```
MAC  MUX_4{TYPE t}=(t:in1 in2 in3 in4, t_mux4:sel)
->
t:
MX_4{t}(in1,in2,in3,in4,ENCODE4_2 sel).

FN AND2 = (bool:a b)
->
bool:BIOP AND.


MAC GNAND2 = (bool:a b)
->
bool:NOT AND2(a,b).

MAC    AND_2 = (t_scratch:in, t_and:sel)
->
t_scratch:
BEGIN
LET     in_s = (I_TO_S{scratch_exp}in)[2],
        sel_s = CAST{bool}sel.
OUTPUT (S_TO_I{scratch_exp}BOOL_STRING{scratch_exp} ([INT j=1..scratch_exp]AND2(BIT_BOOL in_s[j],sel_s) ))[2]
END.


FN XOR = (bool: a b)
->
bool:
CASE (a,b)
OF  (f,f)|(t,t):f
ELSE t
ESAC.
```

```
MAC XOR_B{INT n} = (STRING[n]bit:a b)
->
STRING[n]bit: BIOP XOR.

MAC NOT_B = (STRING[INT n]bit:a)
->
STRING[n]bit:BIOP NOT.

MAC XNOR_B = (STRING[INT n]bit:a b)
->
STRING[n]bit:
NOT_B XOR_B{n}(a,b).

FN AND = (bool: a b)
->
bool:
CASE (a,b)
OF  (t,t):t
ELSE f
ESAC.

MAC DEL{TYPE t} = (t)
->
t:DELAY(?t,1).

#a general dff same as DFF_NO_LOAD#
MAC DFF {TYPE t}=(bool:ck,t_reset:reset,t:in init_value)
->
t:
BEGIN
```

```
MAKE DEL{t}:del.
JOIN in->del.
OUTPUT CASE reset
      OF rst:init_value
      ELSE   del
      ESAC
END.


#a general dff#
MAC DF1 {TYPE t}=(bool:ck,t:in)
->
t:
BEGIN
MAKE DEL{t}:del.
JOIN in->del.
OUTPUT del
END.

#a general latch#
MAC DL1 {TYPE ty}=(bool:ck,ty:in)
->
ty:
BEGIN
MAKE DEL{ty}:del.
JOIN CASE ck
          OF t:in
          ELSE del
          ESAC   ->del.
OUTPUT CASE ck
          OF t:in
```

```
        ELSE del
        ESAC
END.

#a general d latch#
MAC LATCH {TYPE t}=(bool:ck,t_load:load,t:in)
->
t:
BEGIN
MAKE DEL{t}:del.
LET out=CASE load
      OF    write:in
      ELSE  del
      ESAC.
JOIN out->del.
OUTPUT out
END.

#an ACTEL D LATCH#
MAC DLE1D = (bool:ckl loadl,bool:in)
->
bool:#qn#
NOT LATCH{bool}(NOT ckl, CASE loadl
            OF   t:write
            ELSE read
                  ESAC,  in).

MAC PDF1{TYPE t,INT n} = (bool:ck,t_reset:reset,t:in initial_value)
->
t:
     IF n=0 THEN DFF{t}(ck,reset,in,initial_value)
```

```
    ELSE   PDF1{t,n-1}(ck,reset,DFF{t}(ck,reset, in,initial_value),initial_value)
    FI.

#a muxed input dff#
MAC DFM {TYPE ty}=(bool:ck,ty:a b,bool:s)
->
ty:
BEGIN
MAKE DEL{ty}:del.
JOIN CASE s
        OF f:a,
          t:b
        ESAC    ->del.
OUTPUT del
END.
#a resetable DFF, init value is input parameter#
MAC DFF_INIT{TYPE t}=(bool:ck,t_reset:reset,t_load:load,t:in init_value)
->
t:
BEGIN
MAKE DEL{t}:del.
LET out=CASE (load,reset)
      OF   (write,t_reset):in,
           (read,rst):init_value
      ELSE  del
      ESAC.
JOIN out->del.
OUTPUT CASE reset
      OF rst:init_value
      ELSE   del
```

```
    ESAC
END.

#a resetable JKFF, k input is active low#
FN JKFF=(bool:ck,t_reset:reset,bool:j k)
->
bool:
BEGIN
MAKE DEL{bool}:del.
LET out=CASE (j,k,reset)
    OF   (t,t,no_rst):t,
         (f,t,rst):f,
         (t,f,rst):t,
             (f,f,no_rst):f,
             (f,t,no_rst):del,
             (t,f,no_rst):NOT del
    ESAC.
JOIN out->del.
OUTPUT CASE reset
    OF rst:f
    ELSE   del
    ESAC
END.

#a dff resetable non- loadable dff#
MAC DFF_NO_LOAD{TYPE t}=(bool:ck,t_reset:reset,t:in init_value)
->
t:
BEGIN
MAKE DEL{t}:del.
JOIN in->del.
```

```
OUTPUT CASE reset
     OF rst:init_value
     ELSE   del
     ESAC
END.

MAC PDEL{TYPE t,INT n} = (t:in)
->
t:
      IF n=0 THEN DEL{t}in
      ELSE   PDEL{t,n-1} DEL{t} in
      FI.

#the mem control unit for the DWT chip, outputs the memport values for the sparc, and dwt#
#inputs datain from these 2 ports and mux's it to the 2d convolver.#

MAC MEM_CONTROL_NOSCRATCH = (bool:ck,t_reset:reset,t_direction:direction,t_channel:channel,t_octave:octave,
                            t_sparc_addr:sparc_addr_w sparc_addr_r,t_load:zero_hh)

            ->
                            (t_input_mux,t_sparcport,t_dwtport#dwt#):

BEGIN
#the comb. logic for the control of the i/o ports of the chip#
LET ports = (SEQ

        VAR    #defaults, so ? doesnt kill previous mem value#
          rw_sparc:=read,
          rw_dwt:=read,
          cs_dwt:=no_select,
          input_mux:=sparc_in;
```

```
        CASE (direction,octave)
        OF              (forward,oct/0): ( cs_dwt:=select;
                                           input_mux:=dwt_in),

                        (inverse,oct/0):( CASE zero_hh
                                            OF    write:(rw_dwt:=write;
                                                         cs_dwt:=select)
                                            ELSE
                                            ESAC)
        ESAC;

#rw_sparc=write when ck=1 and zero_hh=write, otherwise = read#
        rw_sparc:= CAST{t_load}GNAND2(NOT CAST{bool}zero_hh,ck);

#mux the sparc addr on clock#
#       sparc_addr = GMX4{t_sparc_addr}(sparc_r,sparc_r,sparc_w,sparc_w,ck,f);#

        OUTPUT (input_mux, (sparc_addr_w,sparc_addr_r,rw_sparc),     #sparc port#
                (rw_dwt,cs_dwt)                                     #dwt port#
                )

        ).
OUTPUT ports
END.

# the basic 1d convolver without the control unit#

MAC MULT_ADD = (bool:ck,t_reset: reset,t_input:in, [3]t_and:andsel, [2]t_mux:centermuxsel,[3]t_mux4:muxsel,
                [3]t_and:muxandsel,[4]t_add:addsel, t_direction:direction,[4]t_scratch:pdel)
->
                [4]t_scratch: #pdel are the outputs from the line delays#
```

```
BEGIN

        MAKE   MULTIPLIER:mult,
               [4]ADD_SUB:    add.
#the multiplier outputs#
LET    x3=mult[1],
     x5=mult[2],
     x11=mult[3],
     x19=mult[4],
     x2=mult[5],
     x8=mult[6],
     x30=mult[7],

#the mux outputs#
               mux1=MUX_4{t_scratch}(x11,x5,x8,x2,muxsel[1]),

               mux2=MUX_4{t_scratch}(x19,x30,x8,scratch/0,muxsel[2]),

               mux3=MUX_4{t_scratch}(x11,x5,x8,x2,muxsel[3]),

               centermux=(MUX_2{t_scratch}(pdel[1],pdel[3],centermuxsel[1]),
                    MUX_2{t_scratch}(pdel[2],pdel[4],centermuxsel[2]) ),

               # the AND gates zero the adder inputs every 2nd row#
#the and gate outputs#
               and1=AND_2(pdel[2],andsel[1]),
               and2=AND_2(pdel[3],andsel[1]),
               and3=AND_2(centermux[1],andsel[2]),
               and4=AND_2(centermux[2],andsel[3]),

               add1in=AND_2(mux1,muxandsel[1]),
```

```
            add3in=AND_2(mux3,muxandsel[2]),
            add4in=AND_2(x3,muxandsel[3]).

    JOIN    in        ->mult,
            (and1,add1in,addsel[1]) ->add[1],
            (and3,mux2,addsel[2])   ->add[2],
            (and4,add3in,addsel[3]) ->add[3],
            (and2,add4in,addsel[4]) ->add[4].

    OUTPUT add
END.
# the basic multiplier unit of the convolver #

MAC MULTIPLIER_ST   = (t_input:in)
->
                              [7]t_scratch:
                          #x3,x5,x11,x19,x2,x8,x30#
BEGIN
     MAC INPUT_TO_S{INT n} = (t_input: in)
->
                          (flag,STRING[n]bit): BIOP TRANSFORM_S.
#the multiplier outputs, fast adder code commented out#
   LET   in_s= (INPUT_TO_S{input_exp}in)[2],
      x2=in_s CONC b"0",
      x8=in_s CONC b"000",
      x3 = ADD_S_ACTEL(in_s, x2,b'1),
      x5 = ADD_S_ACTEL(in_s,in_s CONC b"00",b'1 ),
      x11 = ADD_S_ACTEL(x3,x8,b'1),
      x19 = ADD_S_ACTEL(x3,in_s CONC b"0000",b'1),
```

```
        x30=ADD_S_ACTEL(x11,x19,b'1).

LET subsignal = (x2,x8, x3,x5 ,x11,x19,x30).
OUTPUT ((S_TO_I{input_exp+2} x3)[2],(S_TO_I{input_exp+3} x5)[2],(S_TO_I{input_exp+4} x11)[2],
        (S_TO_I{input_exp+5} x19)[2],(S_TO_I{input_exp+1} x2)[2],(S_TO_I{input_exp+3} x8)[2],
        (S_TO_I{input_exp+6} x30)[2])
END.
MAC INBUF{TYPE t} = (t:pad)
->
t:#y#pad.

MAC OBHS{TYPE t} = (t:d)
->
t:#pad#d.

FN CLKBUF = (bool:pad)
->
bool:pad.
#MAC SHIFT{INT p} = (STRING[scratch_exp]bit) ->STRING[scratch_exp+p]bit:BIOP SR_S{p}.#

MAC ADD_S = (STRING[INT m]bit,STRING[INT n]bit)
->
STRING[IF m>=n THEN m+1 ELSE n+1 FI]bit:
BIOP PLUS_S.

MAC INV{INT m} =(STRING[m]bit:a)
->
STRING[m]bit:BIOP NOT.

MAC NEG_S = (STRING[INT n]bit)
->
```

```
STRING[n+1]bit:
BIOP NEGATE_S.

MAC ADD_US = (STRING[INT m]bit,STRING[INT n]bit)
->
STRING[IF m>=n THEN m+1 ELSE n+1 FI]bit:
BIOP PLUS_US.

MAC CARRY= (t_add:in)
->
STRING[1]bit: CASE in
                                        OF add:b"0",
                                         sub:b"1"
                                  ESAC.
#actel adder macros#


#an emulation of a fast ACTEL 16 bit adder with active low carrys#
FN FADD16 = (STRING[scratch_exp]bit: a b,STRING[1]bit:cinb)
          ->
          (STRING[scratch_exp]bit,STRING[1]bit):
BEGIN
LET       a_c =a CONC INV{1}cinb ,
            b_c = b CONC INV{1} cinb ,
          out = ADD_S(a_c,b_c).
OUTPUT(out[2..scratch_exp+1],INV{1} B_TO_S out[1])
END.

#actel 1 bit full adder with active low cin and cout#
MAC FA1B = (bit: ain bin cinb)
          ->
```

```
(bit,bit):#cob,s#
BEGIN
LET     a_c =B_TO_S ain CONC INV{1}B_TO_S cinb ,
        b_c = B_TO_S bin CONC INV{1}B_TO_S cinb ,
      out = ADD_US(a_c,b_c).
OUTPUT(CAST{bit} INV{1} B_TO_S out[1], out[2])
END.


#the actel version of the ADD BIOP's#

MAC ADD_US_ACTEL = (STRING[INT m]bit:ain,STRING[INT n]bit:bin,bit:cinb)
                       ->
                                        STRING[IF m>=n THEN m+1 ELSE n+1 FI]bit:
BEGIN
        MAKE [IF m>=n THEN m ELSE n FI]FA1B:sum.

#unsigned nos so extend by 0#
        LET a_c = IF m>=n THEN ain ELSE ZERO{n-m}b"0" CONC ain FI,
        b_c = IF n>=m THEN bin ELSE  ZERO{m-n}b"0" CONC bin FI.
        LET subsignal = sum.

#lsb#
JOIN    (a_c[IF m>=n THEN m ELSE n FI],b_c[IF m>=n THEN m ELSE n FI],cinb)    ->sum[IF m>=n THEN m ELSE n FI].

FOR INT j=1..(IF m>=n THEN m ELSE n FI) -1
JOIN (a_c[(IF m>=n THEN m ELSE n FI) -j],b_c[(IF m>=n THEN m ELSE n FI) -j], sum[(IF m>=n THEN m ELSE n FI) -j+1][1])
>sum[(IF m>=n THEN m ELSE n FI) -j].

OUTPUT          CAST{STRING[IF m>=n THEN m+1 ELSE n+1 FI]bit}
(INV{1} B_TO_S sum[1][1] CONC
```

```
            CAST{STRING[IF m>=n THEN m ELSE n FI]bit} [INT j=1..IF m>=n THEN m ELSE n FI] sum[j][2])
END.


MAC ADD_S_ACTEL = (STRING[INT m]bit:ain,STRING[INT n]bit:bin,bit:cinb)
                  ->
                                STRING[IF m>=n THEN m+1 ELSE n+1 FI]bit:
BEGIN
   MAKE [IF m>=n THEN m ELSE n FI]FA1B:sum.

#signed nos so sign extend #
 LET a_c = IF m>=n THEN ain ELSE ALL_SAME{n-m}B_TO_S ain[1]  CONC ain FI,
     b_c = IF n>=m THEN bin ELSE ALL_SAME{m-n}B_TO_S bin[1] CONC bin  FI.
      LET subsignal = sum.
#lsb#
JOIN   (a_c[IF m>=n THEN m ELSE n FI],b_c[IF m>=n THEN m ELSE n FI],cinb)    ->sum[IF m>=n THEN m ELSE n FI].

FOR INT j=1..(IF m>=n THEN m ELSE n FI) -1
JOIN (a_c[(IF m>=n THEN m ELSE n FI) -j],b_c[(IF m>=n THEN m ELSE n FI) -j], sum[(IF m>=n THEN m ELSE n FI) -j+1][1])          ->sum[(IF
m>=n THEN m ELSE n FI) -j]].

OUTPUT   CAST{STRING[IF m>=n THEN m+1 ELSE n+1 FI]bit}
(INV{1} B_TO_S sum[1][1] CONC
          CAST{STRING[IF m>=n THEN m ELSE n FI]bit} [INT j=1..IF m>=n THEN m ELSE n FI] sum[j][2])
END.

FN ROUND_BITS = (t_scratch:in,t_round: select)
                                   ->
                                        t_input:
BEGIN
```

```
#THIS ASSUMES THAT THE INPUT_EXP=101111#
        #select chooses a round factor of 3, 4,5#
        #the lsb is the right hand of the string,#
        #the index 1 of the string is the left hand end, &is the msb#
        #so on add ops bit 1 is the carryout#
        LET s1= (I_TO_S{scratch_exp}in)[2],

          msb= B_TO_S s1[1],

          selector =   CASE select         #case conversion for MUX_3#
               OF     shift3:l,
                      shift4:c,
                      shift5:r
               ESAC,

 #needs to be a 16 bit output for the adder#
          shift = MUX_3{STRING[scratch_exp]bit}(
                         msb CONC msb CONC msb CONC s1[1..scratch_exp-3],
                         msb CONC msb CONC msb CONC msb CONC s1[1..(scratch_exp-4)],
                         msb CONC msb CONC msb CONC msb CONC msb CONC s1[1..scratch_exp-5],
                         selector
                                       ),

 #the carry to round, 1/2 value is rounded towards 0#
          cs = CASE select
                    OF shift4: CASE msb
                              OF b"1":s1[scratch_exp-3],        #neg no.#
                               b"0": CASE s1[scratch_exp-3..scratch_exp]
                                    OF  b"1000": b'0          #round down on 1/2 value#
                                    ELSE  s1[scratch_exp-3]
                                    ESAC
```

```
                    ESAC,
        shift3: CASE msb
                    OF b"1":s1[scratch_exp-2],       #neg no.#
                      b"0": CASE s1[scratch_exp-2..scratch_exp]
                           OF  b"100": b'0          #round down on 1/2 value#
                           ELSE  s1[scratch_exp-2]
                           ESAC
                ESAC,

        shift5: CASE msb
                    OF b"1":s1[scratch_exp-4],       #neg no.#
                      b"0": CASE s1[scratch_exp-4..scratch_exp]
                           OF  b"10000": b'0        #round down on 1/2 value#
                           ELSE  s1[scratch_exp-4]
                           ESAC
                    ESAC
           ESAC,

    sum17 =ADD_US_ACTEL(B_TO_S cs, shift,b'1),
    sum = sum17[2..scratch_exp+1],
                         #bit 1 is carry out, gives 16 bit sum#
    subsignal=(cs,sum),
#ACTEL HACK#
    soa = CASE sum[1]
     OF b'1: t,  #saturate to -512#
       b'0: f   #saturate to 512#
     ESAC,

   ss1 = CASE selector
       OF   I:     CASE  sum[4..7]  #these are the 5 msb's form the 13 bit word#
                   OF (b"1111" | b"0000"): t#value in range#
```

```
            ELSE f
            ESAC,

        c:  CASE  sum[5..7]#these are the 3 msb's from the 12 bit word left after#
                     # taking out the 4 sign extension bits#
             OF (b"111" | b"000"): t       #value in range#
            ELSE f
            ESAC,

        r:   CASE  sum[6..7] #these are the 2 msb's from the 11 bit word#
             OF (b"11" | b"00"): t #value in range#
            ELSE f
                           ESAC
          ESAC,
out= MXT{STRING[scratch_exp-6]bit}(b"0111111111",b"1000000000",sum[7..scratch_exp],sum[7..scratch_exp],soa,t,ss1).
      OUTPUT (S_TO_IN out)[2]
END.

MAC LINE_DELAY_ST{TYPE t}=([4]t:in,t_col:wr_address,t_col:rd_address,t_load:rw)
          ->
                  [4]t:
RAM([4]?t).

FN  PR_ADDER_ST = (t_scratch:a b )
->
      t_scratch:
(S_TO_I{scratch_exp} ADD_S((I_TO_S{scratch_exp-1}a)[2],(I_TO_S{scratch_exp-1}b)[2]) ) [2].


FN ADD_SUB_ST = (t_scratch: a b, t_add:sel)
->
```

```
t_scratch:
BEGIN

        LET a_s=(I_TO_S{scratch_exp}a)[2],
          b_s=(I_TO_S{scratch_exp}b)[2],
          sel_bit = CAST{STRING[1]bit}sel,
#ACTEL#
        b_s_inv = XOR_B{scratch_exp}(b_s, ALL_SAME{scratch_exp}sel_bit),

#cinb is active low so cast sel(add->0,sub->1) & invert it#
          out=    ADD_S_ACTEL(a_s ,b_s_inv,CAST{bit}INV{1}sel_bit ),
        binout= out[2..scratch_exp+1].
OUTPUT (S_TO_I{scratch_exp}binout)[2]
END.


MAC ALL_SAME{INT n} = (STRING[1]bit:dummy)
->
STRING[n]bit:
BEGIN
FAULT IF n < 1 THEN "N<1 in ALL_SAME" FI.
OUTPUT IF n=1 THEN dummy
ELSE dummy CONC ALL_SAME{n-1} dummy
FI
END.

MAC CAST {TYPE to} = (TYPE from:in)
->
        to:ALIEN CAST.
```

```
MAC ZERO(INT n) = (STRING[1]bit:dummy)
        ->
STRING[n]bit:
BEGIN
FAULT IF n < 1 THEN "N<1 in ZERO" FI.
OUTPUT IF n=1 THEN b"0"
    ELSE b"0" CONC ZERO(n-1) b"0"
    FI
END.

MAC B_TO_S= (bit:in)
        ->
                STRING[1]bit: CASE in
                        OF b'0:b"0",
                         b'1:b"1"
                    ESAC.

MAC I_TO_S(INT n) = (t_scratch: in)
        ->
                        (flag,STRING[n]bit): BIOP TRANSFORM_S.
MAC S_TO_I(INT n) = (STRING[n]bit:in)
->
 (flag,t_scratch): BIOP TRANSFORM_S.
MAC S_TO_IN = (STRING[input_exp]bit:in)
->
 (flag,t_input): BIOP TRANSFORM_S.
MAC IN_TO_S(INT n) = (t_input: in)
->
 (flag,STRING[n]bit): BIOP TRANSFORM_S.
MAC U_TO_I(INT n) = (STRING[n]bit:in)
->
```

```
(flag,t_scratch): BIOP TRANSFORM_U.

MAC B_TO_I= (bit:in)
->
t_scratch: CASE in
                              OF b'0:scratch/0,
                                 b'1:scratch/1
                        ESAC.

MAC CARRY= (t_add:in)
->
STRING[1]bit: CASE in
                              OF add:b"0",
                                 sub:b"1"
                        ESAC.


MAC BOOL_BIT = (bool:in)
->
STRING[1] bit:
CASE in
OF   t:b"1"
ELSE b"0"
ESAC.
MAC BIT_BOOL= (bit:in)
->
bool:
CASE in
OF   b'1:t
ELSE f
ESAC.
```

```
MAC BOOL_STRING{INT n} = ([n]bool:in)
->
STRING[n] bit:
(LET out = BOOL_BIT in[1].
OUTPUT IF n=1
        THEN out
        ELSE out[1] CONC  BOOL_STRING{n-1}(in[2..n])
        FI
).

#define a few useful gates #
FN NOT = (bool:in) ->bool:
        CASE in
        OF      t:f,
                f:t
        ESAC.
FN MUX = (bool:sel in1 in2) -> bool:
# two input mux, select in1 if sel =t ,otherwise in2 #
        CASE sel
        OF      t:in2,
                f:in1
        ESAC.
FN XNOR=(bool:in1 in2)->bool:
        CASE  (in1,in2)
        OF      (f,f):t,
                (f,t):f,
                (t,f):f,
                (t,t):t
        ESAC.
```

```
FN XOR=(bool:in1 in2)  ->bool:
        CASE   (in1,in2)
        OF       (f,f):f,
                 (f,t):t,
                 (t,f):t,
                 (t,t):f
        ESAC.
FN OR = (bool:in1 in2)  ->bool:
        CASE   (in1,in2)
        OF       (t,bool)|(bool,t):t,
                 (f,f)   : f
        ESAC.
#FN     MYLATCH = (bool:in)    ->bool:DELAY(f,1).#

FN      MYLATCH = (t_reset:reset,bool:in)        ->bool:
BEGIN
MAKE PDEL{bool,0}:del.
LET out = CASE reset
     OF rst:f
     ELSE   del
     ESAC.
JOIN in->del.
OUTPUT out
END.

TYPE t_test = NEW(no|yes).
#..........................................................#
#These functions change types from boolean to inputeger and vice- #
#versa. Supports 1 & 8 bit booleans.                       #
#..........................................................#
```

```
FN INT_BOOL=(t_input:k)        ->bool:            # 1bit input to binary #
                CASE  k
                OF      input/0:f,
                        input/1:t
        ESAC.

FN BOOL_INT=(bool:b) ->t_input:                   # 1 bit bool to input #
                CASE  b
                OF      f:input/0,
                        t:input/1
        ESAC.

FN * =(t_input:a b)      ->t_input: ARITH a*b.
FN % =(t_input:a b)      ->t_input: ARITH a%b.
FN - =(t_input:a b)      ->t_input: ARITH a-b.
FN + =(t_input:a b)      ->t_input: ARITH a+b.
FN = =(t_input:a b)      ->t_test: ARITH IF a=b THEN 2 ELSE 1 FI.

COM
FN CHANGE_SIGN = (t_input:i) ->t_input:         #changes sign for 8-bit 2's#
        ARITH  IF i<0 THEN 128+i         #complement no,  #
                    ELSE i
                    FI.

FN SIGN = (t_input:i)     ->bool:            #gets sign for 2's#
        ARITH  IF i<0 THEN 1         #complement nos  #
                    ELSE 2
                    FI.

FN TEST_SIZE = (t_input:x)      ->bool:
```

```
#tesls to see if the input is bigger than an 8-bit inputeger#
        ARITH IF ( (x<=-128) AND (x>127)) THEN 1
                                                  ELSE 2        FI.


FN INT8_BOOL=(t_input:orig)    ->[8]bool:
BEGIN

        SEQ
                VAR i1:=input/0,          #input variables#
                  i0:=CHANGE_SIGN(orig),
                  b:=(f,f,f,f,f,f,f,SIGN(orig));
        [INT n=1..7]     (
                        i1:=i0%input/2;
                        b[n]:=INT_BOOL(i0-input/2*i1);
                        i0:=i1
                        );
        OUTPUT          CASE TEST_SIZE orig            #checks to see if orig will#
                OF      t:      [8]?bool,        #fit inputo an 8_bit value#
                        f:      b
                ESAC
END.

FN BOOL_INT8=([8]bool:b)     ->t_input:        #converts 8bit boolean to 2's#
BEGIN
        SEQ                                     #complement inputeger #
                VAR     sum:=input/-128 * BOOL_INT(b[8]),
                        exp:=input/1;
                [INT k=1..7]
                        (       sum:=sum+exp*BOOL_INT(b[k]);
                                exp:=input/2 * exp
```

```
                );
        OUTPUT  sum
END.


MOC
FN BOOL_INT10=([10]bool:b)    ->t_input:          #converts 10bit boolean to 2's#
BEGIN
        SEQ                                   #complement integer     #
                VAR     sum:=input/-512 * BOOL_INT(b[10]),
                        exp:=input/1 ;
                [INT k=1..9]
                        (       sum:=sum+exp*BOOL_INT(b[k]);
                                exp:=input/2 * exp
                        );
        OUTPUT  sum
END.
COM
FN BOOL_INT16 =([8]bool:in1 in2)        ->t_input:
# convetrs a 16-bit no., (lsbs,msbs) inputo inputeger form)#
(BOOL_INT8(in1))+((input/256)*BOOL_INT8(in2))+((input/256)*BOOL_INT(in1[8])).
                                                    #hack because of sign extend#
                                                            #of lsb #
MOC
#compute the mean square difference between two arrays of integers#

FN MSE_COLOUR = (t_reset:reset,t_input:a b)    ->[2]t_int32:
BEGIN
        FN SAVE_ERROR = (t_reset:reset,t_int32:diff32) ->t_int32:
        BEGIN
                MAKE PDEL{t_int32,0}:del,
```

- 141 -

```
          PDEL{t_reset,0}:edge.

LET    rising = CASE (reset,edge)
                 OF     (no_rst,rst):diff32,
                        (no_rst,no_rst):del PL diff32
                ELSE    del
                ESAC.
JOIN   rising   ->del,
       reset    ->edge.
OUTPUT          del
END.

MAKE  SAVE_ERROR:save_error.
LET out =(SEQ
       STATE VAR true_count INIT int32/1;
       VAR diff:=int32/0,
       diff32:=int32/0,
       incr:=int32/0;

diff:=CASE reset
       OF rst:int32/0
       ELSE I_32(a) MI I_32(b)
       ESAC;
incr:=CASE reset
       OF rst:int32/0
       ELSE int32/1
       ESAC;
true_count:= CASE reset
          OF rst:int32/1
          ELSE true_count PL  incr
               ESAC;
```

```
        diff32:= (diff TI diff);

OUTPUT (diff32,true_count) ).

JOIN          (reset,out[1])              ->save_error.
OUTPUT        (save_error,save_error DV out[2])
END.


#compute the mean square difference between two arrays of integers#

TYPE t_int32 = NEW int32/(-2147483000..2147483000).
INT period_row=9.

FN I_32 = (t_input:in)      ->t_int32:ARITH in.

FN DV = (t_int32:a b)       ->t_int32:ARITH a%b.
FN PL = (t_int32:a b)       ->t_int32:ARITH a+b.
FN MI = (t_int32:a b)       ->t_int32:ARITH a-b.
FN TI = (t_int32:a b)       ->t_int32:ARITH a*b.


FN MSE_ROW = (t_input:a b)    ->[3]t_int32:
BEGIN SEQ
              STATE VAR err INIT int32/0,
                    count INIT int32/0;
              VAR diff:=int32/0,
              diff32:=int32/0;

        count:=count PL int32/1;
```

```
        diff:=CASE count
                OF int32/(1..period_row):int32/0
                ELSE I_32(a) MI I_32(b)
                ESAC;
        diff32:= (diff TI diff);
        err:=err PL  diff32;
OUTPUT (err, err DV count,count)
END.


FN PRBS10 = (t_reset:reset)      ->[10]bool:
#A 10 bit prbs generator,feedback taps on regs 3 & 10.#
BEGIN
        MAKE   [10]MYLATCH:l,
               XNOR:xnor.

        FOR INT k=1..9 JOIN
               (reset,l[k])         ->l[k+1].

        JOIN   (reset,xnor)         ->l[1],
               (l[10],l[3])         ->xnor.

        OUTPUT         l
END.
FN PRBS11 = (t_reset:reset)      ->[10]bool:
#A 11 bit prbs generator,feedback taps on regs 2 & 11.#
BEGIN
        MAKE   [11]MYLATCH:l,
               XNOR:xnor.
```

```
        FOR INT k=1..10         JOIN
                (reset,l[k])    ->l[k+1].

        JOIN    (reset,xnor)    ->l[1],
                (l[11],l[2])    ->xnor.

        OUTPUT          l[1..10]
END.
COM
FN PRBS16 = (bool:reset)        ->[16]bool:
#A 16 bit prbs generator,feedback taps on regs 1,3,12,16#
BEGIN
        MAKE    [16]MYLATCH:l,
                XOR_4:xor,
                NOT:xnor.

        FOR INT k=1..15         JOIN
                (ck,reset,l[k]) ->l[k+1].

        JOIN    (ck,reset,xnor) ->l[1],
                (l[1],l[3],l[16],l[12])         ->xor,
                xor             ->xnor.
        OUTPUT          ([INT k=1..16]l[k])
END.
FN PRBS12 = (clock:ck,bool:reset)       ->[12]bool:
#A 12 bit prbs generator,feedback taps on regs 1,4,6,12.#
BEGIN
        MAKE    [12]MYLATCH:l,
                XOR_4:xor,
                NOT:xnor.
```

```
        FOR INT k=1..11          JOIN
                (ck,reset,l[k])     ->l[k+1].

        JOIN    (ck,reset,xnor)  ->l[1],
                (l[1],l[4],l[6],l[12])        ->xor,
                xor                 ->xnor.
        OUTPUT          ([INT k=1..12]l[k])
END.

FN PRBS8 = (clock:ck,bool:reset)          ->[8]bool:
#A 8 bit prbs generator,feedback taps on regs 2,3,4,8.#
BEGIN
        MAKE   [8]MYLATCH:l,
                XOR_4:xor,
                NOT:xnor.

        FOR INT k=1..7 JOIN
                (ck,reset,l[k])     ->l[k+1].

        JOIN    (ck,reset,xnor)  ->l[1],
                (l[2],l[3],l[4],l[8])  ->xor,
                xor                 ->xnor.
        OUTPUT          ([INT k=1..8]l[k])
        END.
MOC
#TEST FOR Y U V #
#to test the 2d convolver using prbs input into the forward convolver#
#then outputting to the inverse convolver and checking against the original result#
```

```
FN TEST_COLOUR = (bool:ck,t_reset:reset,bool:extwritel_in csl_in, t_sparc_addr:reg_sel value,t_reset:prbs_reset)
                        ->[3]t_int32:

BEGIN
     FN DEL = (t_load:in)     ->t_load:DELAY(read,1).

     FN PULSE = (t_load:in) ->t_reset:
     CASE (in,DEL in)
     OF      (write,read):rst
     ELSE    no_rst
     ESAC.

     MAKE PRBS11:prbs,
     BOOL_INT10:int_bool,
     DWT:dwt,
     [3]MSE_COLOUR:mse_colour.

JOIN (CASE (prbs_reset,PULSE CASE dwt[3][2]
                         OF write:read,
                          read:write
                         ESAC,PULSE CASE dwt[3][3]
             OF write:read,
              read:write
                         ESAC,PULSE dwt[2][1],PULSE dwt[2][2],PULSE dwt[2][3])

                                             #rerun the prbs at start, or on out of IDWT#
OF   (rst,t_reset,t_reset,t_reset,t_reset,t_reset)|(t_reset,rst,t_reset,t_reset,t_reset,t_reset)|
     (t_reset,t_reset,rst,t_reset,t_reset,t_reset)|(t_reset,t_reset,t_reset,rst,t_reset,t_reset)|
     (t_reset,t_reset,t_reset,t_reset,rst,t_reset)|(t_reset,t_reset,t_reset,t_reset,t_reset,rst):rst
     ELSE  no_rst
     ESAC)                  ->prbs,
```

```
        prbs                ->int_bool,
        (ck,reset,int_bool,extwritel_in,csl_in, reg_sel,value)        ->dwt.

#calcuate the mse error for each channel#
        FOR INT j=1..3 JOIN
        (CASE dwt[2][j]
         OF read:rst
         ELSE no_rst
         ESAC,dwt[1],int_bool)  ->mse_colour[j].
OUTPUT (mse_colour[1][1],mse_colour[2][1],mse_colour[3][1])
END.
FN DWT = (bool,t_reset,t_input,bool,bool,t_sparc_addr:reg_sel value)     ->(t_input,[3]t_load,[3]t_load):IMPORT.
MAC PDEL{TYPE t, INT n} =(t)   ->t:IMPORT.

IMPORTS                dwt/string: DWT_TEST( RENAMED DWT) PDEL.


#TEST FOR LUMINANCE ONLY#
#to test the 2d convolver using prbs input into the forward convolver#
#then outputting to the inverse convolver and checking against the original result#


FN TEST_Y = (bool:ck,t_reset:reset,bool:extwritel_in csl_in, t_sparc_addr:reg_sel value,t_reset:prbs_reset)
                        ->[2]t_int32:

BEGIN
        FN DEL = (t_load:in)     ->t_load:DELAY(read,1).

        FN PULSE = (t_load:in) ->t_reset:
        CASE (in,DEL in)
```

```
        OF      (write,read):rst
        ELSE    no_rst
        ESAC.

        MAKE PRBS11:prbs,
        BOOL_INT10:int_bool,
        DWT:dwt,
        MSE_COLOUR:mse_colour.

JOIN    (CASE (prbs_reset,PULSE dwt[2][1])      #rerun the prbs at start, or on out of IDWT#
        OF      (rst,t_reset)|(t_reset,rst):rst
        ELSE    no_rst
        ESAC)                           ->prbs,

        prbs            ->int_bool,
        (ck,reset,int_bool,extwritel_in,csl_in, reg_sel,value)      ->dwt,
        (CASE dwt[2][1]
         OF  read:rst
         ELSE no_rst
         ESAC,dwt[1],int_bool)   ->mse_colour.
OUTPUT mse_colour
END.
```

APPENDIX B-2

```
#test for abs #

FN ABS_TEST = (STRING[10]bit:in in2)    ->bool: in LE_U in2.
#the state machine to control the address counters#
#only works for 3 octave decomposition in y/2 in u|v#


FN CONTROL_ENABLE = (bool:ck,t_reset:reset,t_channel:new_channel channel,[3]bool:c_blk,STRING[2]bit:subband,
            t_load:load_channel, t_mode:new_mode)

            ->([3]bool#en_blk#,t_octave,[2]bool#tree_done,lpf_block_done#,t_state#reset_state#):
BEGIN
       MAKE DF1{t_state}:state.
#set up initial state thro mux on reset, on HH stay in zz0 state#
LET    start_state = CASE channel
            OFu|v:down1,
                y:up0
            ESAC,
    reset_state= CASE reset
            OF   rst: start_state
            ELSE state
            ESAC.

LET next_values = (SEQ
        VAR en_blk:=[3]f,   #enable blk_count#
            lpf_block_done:=f,        #enable x_count for LPF#
            tree_done:=f,      #enable x_count for other subbands#
            new_state:=reset_state,
            octave:=?t_octave;  #current octave#

     CASE  reset_state
```

```
        ELSE
        ESAC),
    zz1: ( octave:=oct/0;
         en_blk[1]:=t;
         CASE c_blk[1]
         OF     t:(new_state:=zz2;
                en_blk[2]:=t)
         ELSE
         ESAC),
    zz2: ( octave:=oct/0;
         en_blk[1]:=t;
         CASE c_blk[1]
         OF     t:(new_state:=zz3;
                en_blk[2]:=t)
         ELSE
         ESAC),
    zz3: ( octave:=oct/0;
         en_blk[1]:=t;
#now decide the next state, on block(1) carry check the other block carries#
          CASE c_blk[1]
          OF     t:(new_state:=down1;
          en_blk[2]:=t;   #roll over to 0#
          en_blk[3]:=t        #because state zz3 clock 1 pulse#
                 )
          ELSE
          ESAC

     ),
     down1: ( octave:=oct/1;
           en_blk[2]:=t;
           CASE c_blk[2]
```

```
OFup0: ( octave:=oct/2;
        en_blk[3]:=t;
             CASE c_blk[3]
             OF      t:(CASE subband
                     OF b"00":lpf_block_done:=t        #clock x_count for LPF y channel#
                     ELSE  new_state:=up1    #change state when count done#
                     ESAC;

             CASE new_mode      #in luminance & done with that tree#
             OF  stop:tree_done:=t
             ELSE
             ESAC)
             ELSE
             ESAC),
        up1: ( octave:=oct/1;
        en_blk[2]:=t;
             CASE c_blk[2]
             OF      t:(new_state:=zz0;

            CASE new_mode   #in luminance, terminate branch & move to next branch#
            OFstop:(new_state:=down1;
                   en_blk[3]:=t)
            ELSE
            ESAC)
        ELSE
             ESAC),
        zz0: ( octave:=oct/0;
         en_blk[1]:=t;
             CASE c_blk[1]
             OF      t:(new_state:=zz1;
              en_blk[2]:=t)
```

```
        OF     t:(CASE subband
               OF b"00":lpf_block_done:=t       #clock x_count for LPF u|v channel#
               ELSE  new_state:=zz0    #change state when count done#
        ESAC;

        CASE (new_mode,channel)        #stop so finish this tree/branch & move on#
        OF    (stop,u|v):tree_done:=t,
           (stop,y):(en_blk[3]:=t;
               CASE c_blk[3]        #move to next tree#
               OFt:tree_done:=t
               ELSE    new_state:=down1
               ESAC
               )
        ELSE
        ESAC)
          ELSE
          ESAC)

ESAC;

CASE channel
OF u|v:  CASE (c_blk[1],c_blk[2])
             OF     (t,t):tree_done:=t
             ELSE
             ESAC,
    y:  CASE (c_blk[1],c_blk[2],c_blk[3])
             OF     (t,t,t):tree_done:=t
             ELSE
             ESAC
 ESAC;
```

```
#now change to start state if the sequence has finished#
   CASE     tree_done    #in LPF state doesnt change when block done#
   OFt: new_state:= start_state
   ELSE
   ESAC;
#on channel change, use starting state for new channel#
    CASE   load_channel      #in LPF state doesnt change when block done#
    OF     write: new_state:= CASE new_channel
               OFy:up0,
                  u|v:down1
               ESAC
   ELSE
   ESAC;


OUTPUT (new_state,en_blk,octave,(tree_done,lpf_block_done))
).


JOIN  (ck,next_values[1]) ->state.
OUTPUT (next_values[2],next_values[3],next_values[4],reset_state)
END.

FN CHECK = (t_input:x sub size y,t_octave: oct) ->t_sparc_addr:
     ARITH (( x SL 1)+(1 IAND sub) + size*((y SL 1) +(sub SR 1)))SL oct.


#these are the addr gens for the x & y adresses of a pixel given the octave#
#sub&blk no. for each octave.Each x&y address is of the form      #
#   x= count(5 bits){blk(3)..blk(octave+1)}{s} {octave 0's}        #
```

```
#   y= count(5 bits){blk(3)..blk(octave+1)}{s} {octave 0's}        #
#this makes up the 9 bit address for CIF images              #
#the blk & s counters are vertical 2 bit  with the lsb in the x coord  #
#and carry out on 3, last counter is both horiz and vertical counter#
#read_enable enable the block count for the read address, but not the    #
#carry-outs for the mode change, this is done on the write addr cycle #
#by write_enable, so same address values generated on read & write cycles#

FN ADDR_GEN = (bool:ck, t_reset:reset,t_channel:new_channel channel,t_load:load_channel,STRING[2]bit:sub_count,
        STRING[xsize]bit:col_length,STRING[ysize]bit:row_length,STRING[xsize]bit:ximage_string,
        STRING[ysize]bit:yimage_string ,STRING[11]bit:yimage_string_3#yimage*2.5#,
        bool:read_enable write_enable, t_mode:new_mode)

        -> (t_sparc_addr,t_octave,bool#sub finished#,bool#tree_done#,bool#lpf_done#,t_state):
BEGIN
    MAKE COUNTER{xsize-4}:x_count,
        COUNTER{ysize-4}:y_count,
        CONTROL_ENABLE:control,
        [3]BLK_SUB_COUNT:blk_count.

#size of lpf images/2 -1, for y,u|v.  /2 because count in pairs of lpf values #
#lpf same size for all channels!!!#

LET   (x_lpf,y_lpf) = (col_length[1..xsize-4], row_length[1..ysize-4]),

    tree_done = control[3][1],
    lpf_block_done = control[3][2],
    x_en = CASE (tree_done,lpf_block_done)
        OF (t,bool)|(bool,t):t
        ELSE f
        ESAC,
```

```
    blk_en=control[1],
    octave=control[2],

#clk y_count when all blocks done for subs 1-3, or when final blk done for lpf#

    y_en =      CASE sub_count
          OFb"00":CASE (lpf_block_done, x_count[2])
                  OF (t,t):t
                   ELSE f
                   ESAC
           ELSE CASE (tree_done, x_count[2])
               OF (t,t):t
               ELSE f
               ESAC
           ESAC,

    x_msb_out =CASE channel
          OF  y: x_count[1] CONC B_TO_S(blk_count[3][1][2]),      #always the msb bits#
            u|v: b"0" CONC x_count[1]
          ESAC,
    y_msb_out = CASE channel
           OF y:y_count[1] CONC B_TO_S(blk_count[3][1][1]),
            u|v:b"0" CONC y_count[1]
           ESAC,

  x_lsb_out =CASE (octave)        #bit2 is lsb#
     OF (oct/0):([INT k=1..2]blk_count[3-k][1][2])CONC sub_count[2],
        (oct/1):(blk_count[2][1][2], sub_count[2] , b'0),
          (oct/2):sub_count[2] CONC [2]b'0
        ESAC,
```

```
y_lsb_out =CASE (octave)        #bit 1 is msb#
    OF    (oct/0):([INT k=1..2]blk_count[3-k][1][1])CONC sub_count[1],
          (oct/1):(blk_count[2][1][1] , sub_count[1] , b'0),
          (oct/2):sub_count[1] CONC [2]b'0
          ESAC,
  x_addr = x_msb_out CONC BIT_STRING{3}x_lsb_out,
  y_addr = y_msb_out CONC BIT_STRING{3}y_lsb_out,

#enable the sub band counter#
   sub_en = CASE (y_count[2],y_en)
        OF    (t,t):t
       ELSE   f
      ESAC,

   lpf_done = CASE sub_count
         OF   b"00": sub_en           #!!!!CHANGE ACCORDING TO LATENCY IN DECODE#
         ELSE f
         ESAC,

   base_y_sel = CASE channel
        OF   y:l,
         u:c,
         v:r
        ESAC,

   base_rows = MUX_3{STRING[11]bit}(ZERO{11}b"0",b"0" CONC yimage_string[1..ysize]CONC b"0",
                       yimage_string_3,base_y_sel),
 #base address for no of rows for y,u &v memory areas#

   address = x_addr ADD_U ( ((y_addr ADD_U base_rows)[2..12]) MULT_U (CASE channel
             OFy:ximage_string,
```

```
                    u|v:(SR_U{1}ximage_string)[1..xsize]
                ESAC)
           ),

   int_addr = (S_TO_SPARC address)[2].

JOIN    (ck,reset,x_en,x_lpf)          ->x_count,
       (ck,reset,y_en,y_lpf)          ->y_count,
#use new_channel so on channel change control state picks up correct value#
       (ck,reset,new_channel,channel,([INT j=1..3]blk_count[j][2]),sub_count,load_channel,new_mode)
                 ->control.
FOR INT k=1..3 JOIN  (ck,reset,blk_en[k],read_enable OR write_enable,write_enable)       ->blk_count[k].


OUTPUT (int_addr,octave, sub_en,tree_done, lpf_done,control[4])
END.

#a counter to control the sequencing of r/w, token, huffman cycles#
#decide reset is enabled 1 cycle early, and latched to avoid glitches#
#lpf_stop is a is a dummy mode to disable the block writes&huffman data#
#cycles for that block#

FN CONTROL_COUNTER = (bool:ck,t_reset:reset,t_mode:mode new_mode,t_direction:direction)
          ->(t_load,t_cycle,t_reset,bool,bool,t_load,t_cs,t_load,t_cs):

#mode load,cycle,decide reset,read_addr_enable,write_addr_enable,load flags#
#decode write_addr_enable early and latch to avoid feedback loop with pro_mode#
#in MODE_CONTROL#
BEGIN
   MAKE COUNT_SYNC{4}:count.
```

```
LET  count_len = (U_TO_LEN{4} count[1])[2].

LET  out = (SEQ
     VAR  cycle:=skip_cycle,
          decide_reset:= no_rst,
          load_mode:=read,
          load_flags:=read,
          cs_new:=no_select,
          cs_old:=select,
          rw_old:=read,
          read_addr_enable:=f,
          write_addr_enable:=f;

 CASE direction
 OFforward: CASE mode
      OF send|still_send|ipf_send:  CASE   count_len
                          OF   len/(0..3):(read_addr_enable:=t;
                                    cs_new:=select),
                            len/(4):(cycle:=token_cycle;
                              load_flags:=write;
                              write_addr_enable:=t),

                            len/(5..7) : (write_addr_enable:=t;
                              CASE new_mode
                              OFstop|ipf_stop:(cycle:=skip_cycle;
                                        rw_old:=read;
                                        cs_old:=no_select),
                                 vold:(cycle:=skip_cycle;
                                   rw_old:=write)
                              ELSE (cycle:=data_cycle;
                                          rw_old:=write)
```

```
            ESAC),
         len/8:(decide_reset:=rst;
            CASE new_mode
                        OF      stop|lpf_stop:(cycle:=skip_cycle;
                                        rw_old:=read;
                                        cs_old:=no_select),
               void:(cycle:=skip_cycle;
                          load_mode:=write;
                               rw_old:=write)

                        ELSE    (cycle:=data_cycle;
                        load_mode:=write;
                            rw_old:=write)
                        ESAC)
        ELSE
        ESAC,

still:        CASE   count_len
                OF     len/(0..3):(read_addr_enable:=t;
                                   cs_new:=select),
                     len/(4):(cycle:=token_cycle;
                         write_addr_enable:=t;
                         load_flags:=write),
                     len/(5..7):( rw_old:=write;
                                   write_addr_enable:=t;
                        CASE new_mode
                         OF     void_still:cycle:=skip_cycle
                         ELSE   cycle:=data_cycle
                         ESAC),

                     len/8:(decide_reset:=rst;
```

```
                    rw_old:=write;
                    load_mode:=write;
                     CASE new_mode
                      OF    void_still:cycle:=skip_cycle
                      ELSE   cycle:=data_cycle
                     ESAC)

                ELSE
                ESAC,

lpf_still:       CASE   count_len
                OF    len/(0..3):(read_addr_enable:=t;
                                 cs_new:=select),
                   len/(4):(cycle:=token_cycle;
                            write_addr_enable:=t;
                            load_flags:=write),
                   len/(5..7):(cycle:=data_cycle;
                                 rw_old:=write;
                                 write_addr_enable:=t),
                   len/8:( cycle:=data_cycle;
                            rw_old:=write;
                            decide_reset:=rst;
                            load_mode:=write)
                 ELSE
                 ESAC,

void:        CASE count_len
                 OF   len/(0..3):(read_addr_enable:=t;
                cs_new:=select),
             len/4:(load_flags:=write;
                 cycle:=token_cycle;  #dummy token cycle for mode update#
```

```
write_addr_enable:=t),
          len/(5..7):(write_addr_enable:=t; #keep counters going#
 CASE new_mode
                 OF     stop:(rw_old:=read;
                        cs_old:=no_select)
 ELSE rw_old:=write
 ESAC),
        len/8:( decide_reset:=rst;
CASE new_mode
             OF     stop:(rw_old:=read;
                         cs_old:=no_select)
ELSE    (load_mode:=write;
                   rw_old:=write)
ESAC)

        ELSE
        ESAC,

void_still:   CASE  count_len
                 OF   len/0: write_addr_enable:=t,    #allow for delay#
              len/(1..3):(write_addr_enable:=t;
                  rw_old:=write),
                     len/4:(rw_old:=write;
                          load_mode:=write;
                decide_reset:=rst)
                 ELSE
                 ESAC
  ELSE
  ESAC,

inverse: CASE mode
```

```
OF send|still_send|lpf_send: CASE count_len
                    OF   len/(0..3):(read_addr_enable:=t),
                      len/(4):(cycle:=token_cycle;
                         write_addr_enable:=t;
                         load_flags:=write),
                      len/(5..7) : (write_addr_enable:=t;
                         CASE new_mode
                         OF     stop|lpf_stop:(cycle:=skip_cycle;
                                        rw_old:=read;
                                        cs_old:=no_select),
                            void:(cycle:=skip_cycle;
                               rw_old:=write)
                         ELSE   (cycle:=data_cycle;
                            rw_old:=write)
                         ESAC),
                      len/8:(decide_reset:=rst;
                         CASE new_mode
                         OF     stop|lpf_stop:(cycle:=skip_cycle;
                                        rw_old:=read;
                                        cs_old:=no_select),
                            void:(cycle:=skip_cycle;
                               load_mode:=write;
                               rw_old:=write)

                         ELSE   (cycle:=data_cycle;
                            load_mode:=write;
                            rw_old:=write)
                         ESAC)

                   ELSE
                   ESAC,
```

```
still:       CASE count_len
                OF  len/(0):,        #skip to allow reset in huffman#
        len/(1):(cycle:=token_cycle;
                        write_addr_enable:=t),
        len/(2..4):(rw_old:=write;
                        write_addr_enable:=t;
             CASE new_mode
             OFvoid_still:cycle:=skip_cycle
             ELSE cycle:=data_cycle
             ESAC),

                  len/5:( rw_old:=write;
                        decide_reset:=rst;
             load_mode:=write;
             CASE new_mode
                        OF    void_still:cycle:=skip_cycle
                        ELSE   cycle:=data_cycle
                        ESAC)
                  ELSE
                  ESAC,
lpf_still:   CASE count_len
                OF  len/(0):,        #match with previous#
        len/(1):(         #skip for write enb delay#
                        write_addr_enable:=t),
                  len/(2..4):(cycle:=data_cycle;
                        rw_old:=write;
                        write_addr_enable:=t),
                  len/5:(cycle:=data_cycle;
                        rw_old:=write;
                        decide_reset:=rst;
                        load_mode:=write)
```

```
                ELSE
                ESAC,
void:           CASE count_len
                OF   len/(0..3):(read_addr_enable:=t),
                        len/4:(load_flags:=write;
            cycle:=token_cycle;  #dummy token cycle for mode update#
                                write_addr_enable:=t),
                        len/(5..7):(write_addr_enable:=t;
              CASE new_mode
                                OF      stop:(rw_old:=read;
                                           cs_old:=no_select)
                                ELSE rw_old:=write
                                ESAC),
                        len/8:( decide_reset:=rst;
                               CASE new_mode
                               OF      stop:(rw_old:=read;
                                          cs_old:=no_select)
                               ELSE    (load_mode:=write;
                                   rw_old:=write)
                               ESAC)
                ELSE
                ESAC,
void_still:      CASE  count_len
                 OF  len/(0):,           #match with rest#
           len/1:write_addr_enable:=t, #dummy as write delayed#
                       len/(2..4):(write_addr_enable:=t;
                                rw_old:=write),
                       len/5:  (rw_old:=write;
             load_mode:=write;
             decide_reset:=rst)
                 ELSE
```

```
                        ESAC
        ELSE
        ESAC
ESAC;

  OUTPUT (load_mode,cycle,DF1{t_reset}(ck,decide_reset),read_addr_enable,
     DFF{bool}(ck,reset,write_addr_enable,f),load_flags,
     cs_new,rw_old,cs_old)
).

JOIN  (ck,CASE reset
       OF   rst:rst
       ELSE out[3]
       ESAC,t)      ->count.

OUTPUT out
END.


#..........................................................#
#A set of boolean ,ie gate level counters                #
#..........................................................#

#..........................................................#
#The basic toggle flip-flop plus and gate for a synchronous counter  #
#input t is the toggle ,outputs are q and tc (toggle for next counter#
#stage                                         #
#..........................................................#

MAC  BASIC_COUNT = (bool:ck ,t_reset:reset,bool: tog)    ->(STRING[1]bit,bool):
```

```
BEGIN
    MAKE DFF{bool}:dlat,
        XOR   :xor,
        AND   :and.

    JOIN  (ck,reset,xor,f)->dlat,
        (dlat,tog)   ->and,
        (tog,dlat)   ->xor.
    OUTPUT (CAST{STRING[1]bit} dlat,and)
END.

#..................................................................#
# The n-bit macro counter generator, en is the enable, the outputs   #
#are msb(bit 1).....lsb,carry.This is the same order as ELLA strings are stored#
#..................................................................#

MAC COUNT_SYNC{INT n} = (bool:ck,t_reset: reset,bool: en )->(STRING[n]bit,bool):
(LET out = BASIC_COUNT(ck,reset,en) .

  OUTPUT        (   IF n=1
             THEN (out[1],out[2])
             ELSE ( LET outn = COUNT_SYNC{n-1}(ck,reset,out[2]) .
             OUTPUT (outn[1] CONC out[1],outn[2])
             )
             FI)
 ).
 COM
 FN TEST_COUNT_SYNC = (bool:ck,t_reset: reset,bool: en ) ->([4]bool,bool):
     COUNT_SYNC{4}(ck,reset,en).
 MOC
```

```
#..............................................................#
#The basic toggle flip-flop plus and gate for a synchronous counter   #
#input t is the toggle, updown detms the direction ,outputs are q and #
# tc (toggle for next counterstage, active low for down/high for up)  #
#..............................................................#

MAC  BASIC_COUNT_UD = (bool:ck ,t_reset:reset,bool: tog, t_updown:updown)   ->[2]bool:

BEGIN
            MAKE DFF{bool}:dlat.
    LET     toggle = tog,
     xorn = CASE updown
          OF   up: CASE (toggle,dlat) #xor#
               OF    (t,t)|(f,f):f
               ELSE t
               ESAC,
            down:CASE (toggle,dlat) #xnor#
                    OF    (t,t)|(f,f):t
                    ELSE f
                    ESAC
          ESAC,
     cout = CASE updown
         OFup:CASE (dlat,toggle)  #AND#
                OF (t,t):t
                ELSE  f
                ESAC,
            down:CASE (dlat,toggle)   #OR#
                OF(f,f):f
                ELSE   t
                ESAC
```

```
        ESAC.

          JOIN  (ck,reset,xorn,f)->dlat.
          OUTPUT (dlat,cout)
END.

#..............................................................#
# The n-bit macro u/d counter generator, en is the enable, the outputs   #
#are msb(bit 1).....lsb,carry.This is the same order as ELLA strings are stored#
#first enable is active low on down, so invert.                 #
#..............................................................#
MAC COUNT_SYNC_UD{INT n} = (bool:ck,t_reset: reset,bool:en, t_updown:updown)     ->(STRING[n]bit,bool):
BEGIN
   MAKE [n]BASIC_COUNT_UD:basic_count.
   LET enable = ([INT k=1..n-1] basic_count[k+1][2]) CONC CASE updown       #invert enable if down count#
                                          OF  up:en
                                         ELSE NOT en
                                         ESAC .
   FOR INT k=1..n JOIN  (ck,reset,enable[k],updown)   ->basic_count[k].
OUTPUT (BOOL_STRING{n}([INT k=1..n]basic_count[k][1]), basic_count[1][2])
END.

COM
FN TEST_COUNT_SYNC_UD = (bool:ck,t_reset: reset,bool: en,t_updown:updown ) ->([4]bool,bool):
          COUNT_SYNC_UD{4}(ck,reset,en,updown).
MOC


#the basic x/y counter, carry out 1 cycle before final count given by x_lpf/y_lpf#
MAC COUNTER{INT n} = (bool:ck,t_reset:reset,bool:en,STRING[n]bit:x_lpf)  ->(STRING[n]bit,bool):
BEGIN
```

```
MAKE     COUNT_SYNC{n}:x_count.

LET out =  x_count[1],
   final_count = out EQ_U x_lpf,
   final_count_en=CASE (final_count,en)
          OF   (t,t):t
          ELSE f
          ESAC,
#reset after 4 counts at final count value#
    cnt_reset = CASE reset
        OF    rst:rst
        ELSE CASE DF1{bool}(ck,final_count_en) #reset taken out of DFF 12/6#
            OFt:rst
            ELSE no_rst
            ESAC
        ESAC.
 JOIN  (ck,cnt_reset,en)    ->x_count.
 OUTPUT (out,final_count)
 END.
 COM
 #the basic y  counter, carry out 1 cycle before final count given by y_lpf#
 #reset at end of channel given by system reset                   #
 MAC Y_COUNTER = (bool:ck,t_reset:reset,bool:en,STRING[4]bit:y_lpf)  ->(STRING[4]bit,bool):
 BEGIN

 MAKE    COUNT_SYNC{4}:y_count.

 LET out =  y_count[1].
 JOIN    (ck,reset,en)   ->y_count.
 OUTPUT (out, out EQ_U y_lpf)
```

```
END.
MOC


COM
#the blk, or sub-band counters, carry out on 3#
FN BLK_SUB_COUNT = (bool:ck,t_reset:reset, bool:en)         ->(STRING[2]bit,bool):
BEGIN
   MAKE COUNT_SYNC{2}:blk_count.
   LET out = blk_count[1].
   JOIN (ck,reset,en)  ->blk_count.
OUTPUT(out,out EQ_U (C_TO_S{2}col/3)[2])
END.
MOC

#the blk, or sub-band counters, carry out on 3, cout_en enables the carry out, & cin_en AND en  enables the count#
FN BLK_SUB_COUNT = (bool:ck,t_reset:reset, bool:en cin_en cout_en)          ->(STRING[2]bit,bool):
BEGIN
     MAKE COUNT_SYNC{2}:blk_count.
     LET out = blk_count[1].
     JOIN (ck,reset,en AND cin_en)      ->blk_count.
OUTPUT(out,(out EQ_U (C_TO_S{2}col/3)[2]) AND cout_en)
END.


FN LAST_BLK_COUNT = (bool:ck,t_reset:reset, bool:en,t_channel:channel,bool:line_finished) ->
                              (STRING[2]bit,[2]bool#x_en,y_en#):
BEGIN
   MAKE BASIC_COUNT : lsb msb.
JOIN  (ck,reset,en) ->lsb,
   (ck,reset, CASE channel
```

```
          OF y:lsb[2],
           u|v:line_finished
          ESAC) ->msb.
LET out = (msb[1]CONClsb[1]).
OUTPUT (out, CASE channel
         OF y:(out EQ_U (C_TO_S{2}col/3)[2],line_finished),
        u|v:(lsb[2],msb[2])
         ESAC)
END.
#the L1 norm calculator/ comparison constants& flag values#
#adding 4 absolute data values so result can grow by 2 bits#
#5 cycle sequence, a reset cycle with no data input, followed#
#by 4 data cycles#

MAC L1NORM = (bool:ck, t_reset:reset, STRING[INT n]bit:in) ->STRING[n+2]bit:
BEGIN
   MAKE DF1{STRING[n+4]bit}:in2.


LET in_s =in,
   msb = ALL_SAME{n}(B_TO_Sin_s[1]),
COM
   add_in1 = in2 CONC in_s[1],  #in_s[1] is the carryin to the adder#
                               #,lsb so gen carry to next bit#
   add_in2 = ((in_s XOR_B msb)CONC in_s[1]),
   #adder=ADD_U(add_in1,add_in2),#
MOC
   add_in1 = (in_s XOR_B msb),
   rst_mux = CASE reset
        OF      rst:ZERO{n+4}b"0"
        ELSE in2
```

```
        ESAC,

   adder=ADD_US_ACTEL(add_in1,rst_mux,CASE in_s[1]
                 OF b'1:b'0
                 ELSE b'1
                 ESAC ),
   out =adder[2..(n+5)].

JOIN  (ck,out)   ->in2.

OUTPUT in2[3..n+4]
END.

#the block to decide if all its inputs are all 0#
FN ALL_ZERO = (bool:ck, t_reset:reset, t_input:in)   ->bool:
BEGIN

   MAKE DF1{bool}:out.

LET in_s =(IN_TO_S{input_exp}in)[2],

   in_eq_0 = in_s EQ_U ZERO{input_exp}b"0",  #in =0#
#1 if reset high, & OR with previous flag#
   all_eq_0 = CASE reset
        OF rst:   in_eq_0
        ELSE CASE out
          OF   t:f
          ELSE in_eq_0
          ESAC
        ESAC.
```

```
JOIN  (ck,all_eq_0)->out.
OUTPUT out
END.

MAC ABS_NORM = (bool:ck, t_reset:reset,STRING[result_exp-2]bit:qshift, STRING[INT n]bit:in)
                          ->(STRING[n+2]bit,bool#all <qshift#):
BEGIN
    MAKE DF1{STRING[n+4]bit}:in2,
      DF1{bool}:out.
LET   abs_in = ABS_S in,
   rst_mux = CASE reset
         OF      rst:ZERO{n+4}b"0"
         ELSE  in2
         ESAC,

   adder = ADD_US_ACTEL(abs_in,rst_mux,b'1),
   add_s =adder[2..(n+5)],
   in_small = abs_in LT_U qshift,
 #1 if reset high, & OR with previous flag#
     all_small =  CASE reset
            OF  rst:  t
            ELSE CASE in_small
          OF    f:f
          ELSE out
          ESAC
            ESAC.

 JOIN    (ck,add_s)    ->in2,
     (ck,all_small)   ->out.

 OUTPUT (in2[3..n+4],out)
```

```
END.


#the decide fn block#
FN DECIDE = (bool:ck,t_reset:reset,t_result:q_int,t_input:new old, t_result: threshold comparison,
          t_octave:octs,t_load:load_flags)      ->[7]bool:
#nzflag,origin,noflag,ozflag,motion,pro_new_z,pro_no_z#
BEGIN
   MAKEL1NORM{input_exp}: oz,
      ABS_NORM{input_exp}: nz,
      ABS_NORM{input_exp+1}:no,
      LATCH{[7]bool}:flags.

 LET   qshift=(I_TO_SC{result_exp}q_int)[2][1..result_exp-2],
 #divide by 4 as test is on coeff values not block values#

   n_o =(IN_TO_S{input_exp}new)[2] SUB_S  (IN_TO_S{input_exp}old)[2], #new-old,use from quant#
   nzflag = nz[1] LE_U (I_TO_SC{result_exp}threshold)[2],      #delay tests for pipelined data#
   noflag = no[1] LE_U (I_TO_SC{result_exp}comparison)[2],
   ozflag = oz EQ_U ZERO{input_exp}b"0",
   origin = nz[1] LE_U no[1],
   nz_plus_oz = nz[1] ADD_U oz,

   pro_new_z = nz[2],

   pro_no_z = no[2],

   shift_add_sel = CASE DF1{t_octave}(ck,octs)          #delay octs to match pipelin delay#
            OF     oct/0:uno,
```

```
                    oct/1:dos,
                    oct/2:tres,
                    oct/3:quatro
                ESAC,
#keep 13 bits here to match no, keep msb's#
    shift_add= MUX_4{STRING[input_exp+3]bit}(          #delay octs to match pipelin delay#
                        nz_plus_oz[1..input_exp+3],
                        b"0"CONC nz_plus_oz[1..input_exp+2],
                        b"00"CONC nz_plus_oz[1..input_exp+1],
                        b"000"CONC nz_plus_oz[1..input_exp],
                        shift_add_sel
                        ),

    motion = shift_add LE_U no[1],
#value for simulation#
    nz_r = (SC_TO_I{12} nz[1])[2],
    no_r = (SC_TO_I{13} no[1])[2],
    oz_r = (SC_TO_I{12} oz)[2],
    sa_r = (SC_TO_I{13} shift_add)[2].


JOIN  (ck,reset,qshift,(IN_TO_S{input_exp}new)[2])  ->nz,
   (load_flags,(nzflag,origin,noflag,ozflag,motion,pro_new_z,pro_no_z)) ->flags,
   (ck,reset,qshift,CAST{ STRING[input_exp+1]bit}n_o) ->no,
   (ck,reset,(IN_TO_S{input_exp}old)[2])  ->oz.
OUTPUT flags
END.
#the buffer for the FIFO#

#a pulse generator, glitch free#
FN PULSE = (bool:ck,t_reset:reset,t_load:in)  ->t_load:
```

```
CASE (in,DFF{t_load}(ck,reset,in,read))
OF     (write,read):write
ELSE   read
ESAC.

#the length of the huffman encoded word#
FN LENGTH = (t_input:mag_out)        ->STRING[5]bit:
CASE mag_out          #length of inputoded word#
OF     input/0:b"00001",
      input/1:b"00011",
      input/2:b"00100",
      input/3:b"00101",
      input/4:b"00110",
      input/5:b"00111",
      input/6:b"01000",
      input/(7..21):b"01100"
 ELSE              b"10000"
 #      input/(22..37):b"10000"#
 ESAC.

 FN REV_BITS = (STRING[8]bit:in)   ->STRING[8]bit:CAST{STRING[8]bit}(in[8],in[7],in[6],in[5],in[4],in[3],in[2],in[1]).

 FN FIFO_BUFFER = (bool:ck, t_reset:reset,t_direction:direction,t_cycle:cycle,t_mode:mode,
        t_input:value mag_out_huff, STRING[16]bit:fifo_in,t_fifo:fifo_full fifo_empty,
         STRING[32]bit:shift,STRING[2]bit:token_length, bool:flush_buffer,t_quant:ipf_quant)

        ->(STRING[16]bit,STRING[16]bit,STRING[16]bit,STRING[5]bit,t_load,t_load):
 #fifo_out, s, fifo_read fifo_write#

 BEGIN
    MAKEDFF_INIT{STRING[16]bit}:low_word high_word,
```

```
    DFF_INIT{STRING[5]bit}:s,
    DFF_INIT{t_high_low}:high_low,
    MUX_2{STRING[16]bit}:high_in low_in high_out low_out.


LET  dir_sel =     CASE direction
           OF    forward:left
           ELSE  right
           ESAC,

  length = CASE cycle
         OF   token_cycle:b"000" CONC token_length,
              skip_cycle:b"00000",
              data_cycle:   CASE mode  #on LPF_STILL length fixed, given by input_exp-shift const#
                            OF    lpf_still:((LEN_TO_U{5} len/input_exp)[2] SUB_U
                                  (Q_TO_U{3} lpf_quant)[2])[2..6]
                            ELSE  LENGTH MUX_2{t_input}(value,mag_out_huff,dir_sel)
                            ESAC
         ESAC,

   select_s = CASE direction
         OF    forward:b"0" CONC s[2..5]
           ELSE  s
         ESAC,

    new_s = (ADD_US_ACTEL(select_s,length,b'1))[2..6],      #6 bits#
 #if new s pointer >16#
 #on inverse passed first 16 bits, active from [16,31] #

     high_low_flag = new_s GE_U b"10000",
```

```
#forward#
   fifo_not_full = CASE    fifo_full
                   OF      ok_fifo: write
                   ELSE         read
                   ESAC,


    fifo_write = CASE high_low      #type change#
                 OF  high:write
                 ELSE     CASE flush_buffer            #flush buffer when frame finished#
                          OF     t:write               #needs 2 cycles to clear#
                          ELSE   CASE DFF{bool}(ck,reset,flush_buffer,f)
                                 OF     t:write
                                 ELSE   read
                                 ESAC
                          ESAC
                 ESAC,
 #from inverse#

   data_ready =   CASE fifo_empty
                  OF     ok_fifo:write
                  ELSE   read
                  ESAC,


    load_low      =     CASE reset            #load low on reset to start things#
                    OF     rst:write,
                           no_rst: PULSE(ck,reset, CASE (high_low_flag,data_ready) #load low word#
                                OF     (t,write):write
                                ELSE   read
                                ESAC)
```

```
            ELSE read
            ESAC,
#delay reset for s and load_high#
    reset_s =     DFF{t_reset}(ck,reset,reset,rst),

    load_high     =CASE reset_s    #load high next#
              OF     rst:write,
                  no_rst: PULSE(ck,reset, CASE (high_low_flag,data_ready) #load high word#
                      OF      (f,write):write
                      ELSE    read
                      ESAC)
              ELSE read
              ESAC,

    fifo_read     = CASE load_low       #read control for data_in FIFO#
               OF   write:read
               ELSE  CASE load_high
                  OF     write:read
                  ELSE   write
                  ESAC
               ESAC,

 #control signals#

 (write_low,write_high) =CASE direction
               OF     forward:[2]fifo_not_full
               ELSE   (load_low,load_high)
               ESAC,

 (high_out_sel,low_out_sel) =  CASE direction
                   OF     forward:CASE high_low
```

```
                OF     high:(left,right)
                ELSE   (right,left)
                ESAC

           ELSE   [2]CAST{t_mux}(s GE_U b"10000")
           ESAC.

JOIN
    (shift[17..32],fifo_in,dir_sel)           ->high_in ,

    (shift[1..16],fifo_in,dir_sel)            ->low_in,

    (high_word,low_word,high_out_sel)         ->high_out,

    (low_word,high_word,low_out_sel)          ->low_out,

    (ck,reset,write_low,low_in,ZERO{16}b"0")  ->low_word,

    (ck,reset,write_high,high_in,ZERO{16}b"0")   ->high_word,

    (ck,reset,fifo_not_full,CASE high_low_flag
              OF    t:high
              ELSE  low
              ESAC,low)          ->high_low,

    (ck,CASE forward
       OFforward:reset
       ELSE reset_s
       ESAC,CASE direction
          OF  forward:fifo_not_full
```

```
        ELSE data_ready
        ESAC,       new_s,ZERO{5}b"0")   ->s.

OUTPUT (low_word,low_out,high_out,s,fifo_read,fifo_write)
END.
#the HUFFMAN decode/encode function#

#a pulse generator, glitch free#

FN PULSE = (bool:ck,t_reset:reset,t_load:in)  ->t_load:
 CASE (in,DFF{t_load}(ck,reset,in,read))
 OF    (write,read):write
 ELSE   read
 ESAC.


FN SHIFT32_16 = (STRING[32]bit:buffer,STRING[5]bit:s)     ->STRING[16]bit:
#left justified value, s shift const#
BEGIN
LET shift = (s AND_B b"01111")[2..5].     #input values rotated so always shift<16#
OUTPUT
CAST{STRING[16]bit}([INT j=1..16] MX16(CAST{STRING[16]bit}([INT i=1..16]buffer[j-1+i]),shift) )
END.

FN SHIFT16X16_32 = (STRING[16]bit:o n, STRING[4]bit:sel)  ->STRING[32]bit:
BEGIN
LET sel_mux4= CASE sel[1..2]
        OFb"00":sel[3..4]
        ELSE b"11"
```

```
        ESAC,
sel_mux4_high = CASE sel[1..2]
            OF      b"11":sel[3..4]
            ELSE    b"00"
            ESAC,
sel_mux8 =  CASE sel[1]
        OF b'0: sel[2..4]
        ELSE b"111"
        ESAC,
sel_mux8_high =  CASE sel[1]
            OF b'1: sel[2..4]
            ELSE b"000"
            ESAC.
OUTPUT CAST{STRING[32]bit}(
 MX_4{bit}(n[1],o[1],o[1],o[1],CAST{[2]bool}sel_mux4),
 MX_4{bit}(n[2],n[1],o[2],o[2],CAST{[2]bool}sel_mux4),
 MX_4{bit}(n[3],n[2],n[1],o[3],CAST{[2]bool}sel_mux4),

 MUX_8{bit}(n[4],n[3],n[2],n[1],o[4],o[4],o[4],o[4],CAST{[3]bool}sel_mux8),
 MUX_8{bit}(n[5],n[4],n[3],n[2],n[1],o[5],o[5],o[5],CAST{[3]bool}sel_mux8),
 MUX_8{bit}(n[6],n[5],n[4],n[3],n[2],n[1],o[6],o[6],CAST{[3]bool}sel_mux8),
 MUX_8{bit}(n[7],n[6],n[5],n[4],n[3],n[2],n[1],o[7],CAST{[3]bool}sel_mux8),

 MX16(CAST{STRING[8]bit}(([INT i=1..8]n[9-i]) CONC ALL_SAME{8}B_TO_S o[8],sel[1..4]),
 MX16(CAST{STRING[9]bit}([INT i=1..9]n[10-i]) CONC ALL_SAME{7}B_TO_S o[9],sel[1..4]),
 MX16(CAST{STRING[10]bit}([INT i=1..10]n[11-i]) CONC ALL_SAME{6}B_TO_S o[10],sel[1..4]),
 MX16(CAST{STRING[11]bit}([INT i=1..11]n[12-i]) CONC ALL_SAME{5}B_TO_S o[11],sel[1..4]),

 MX16(CAST{STRING[12]bit}([INT i=1..12]n[13-i]) CONC ALL_SAME{4}B_TO_S o[12],sel[1..4]),
 MX16(CAST{STRING[13]bit}([INT i=1..13]n[14-i]) CONC ALL_SAME{3}B_TO_S o[13],sel[1..4]),
 MX16(CAST{STRING[14]bit}([INT i=1..14]n[15-i])CONC ALL_SAME{2}B_TO_S o[14],sel[1..4]),
```

```
MX16(CAST{STRING[16]bit}(([INT i=1..15]n[16-i])CONC o[15]),sel[1..4]),

MX16(CAST{STRING[16]bit}([INT i=1..16]n[17-i]),sel[1..4]),

MX16(CAST{STRING[16]bit}(b'0 CONC ([INT i=1..15]n[17-i])),sel[1..4]),
MX16(ZERO{2}b"0" CONC CAST{STRING[14]bit}([INT i=1..14]n[17-i]),sel[1..4]),
MX16(ZERO{3}b"0" CONC CAST{STRING[13]bit}([INT i=1..13]n[17-i]),sel[1..4]),
MX16(ZERO{4}b"0" CONC CAST{STRING[12]bit}([INT i=1..12]n[17-i]),sel[1..4]),

MX16(ZERO{5}b"0" CONC CAST{STRING[11]bit}([INT i=1..11]n[17-i]),sel[1..4]),
MX16(ZERO{6}b"0" CONC CAST{STRING[10]bit}([INT i=1..10]n[17-i]),sel[1..4]),
MX16(ZERO{7}b"0" CONC CAST{STRING[9]bit}([INT i=1..9]n[17-i]),sel[1..4]),
MX16(ZERO{8}b"0" CONC CAST{STRING[8]bit}([INT i=1..8]n[17-i]),sel[1..4]),

MUX_8{bit}(b'0,n[16],n[15],n[14],n[13],n[12],n[11],n[10],CAST{[3]bool}sel_mux8_high),
MUX_8{bit}(b'0,b'0,n[16],n[15],n[14],n[13],n[12],n[11],CAST{[3]bool}sel_mux8_high),
MUX_8{bit}(b'0,b'0,b'0,n[16],n[15],n[14],n[13],n[12],CAST{[3]bool}sel_mux8_high),
MUX_8{bit}(b'0,b'0,b'0,b'0,n[16],n[15],n[14],n[13],CAST{[3]bool}sel_mux8_high),

MX_4{bit}(b'0,n[16],n[15],n[14],CAST{[2]bool}sel_mux4_high),
MX_4{bit}(b'0,b'0,n[16],n[15],CAST{[2]bool}sel_mux4_high),
MX_4{bit}(b'0,b'0,b'0,n[16],CAST{[2]bool}sel_mux4_high),

b'0
   )
END.

MAC REV_4 = (STRING[4]bit:in)        ->STRING[4]bit:CAST{STRING[4]bit}((in[4],in[3],in[2],in[1])).

#in is data from bus, fifo_empty is input fifo control#
FN HUFFMAN_DECODE =(t_mode:mode,STRING[2]bit:token_length_in,STRING[32]bit:buffer,STRING[5]bit:s)
```

```
            ->(bit,t_input,STRING[2]bit#token#):
BEGIN

   MAKE SHIFT32_16:input_decode.
COM
LET    mag_out2 = CASE input_decode[9..12]
          OFb"1111":(input_decode[13..16] ADD_U b"10110")#add 22 to give value#
          ELSE input_decode[9..12] ADD_U b"00111"       #add 7 to give value#
          ESAC,
MOC
LET    sel_9_12 =    CASE input_decode[9..12]
               OF     b"1111":t
          ELSE f
               ESAC,
   mag_out2 = CASE sel_9_12
         OF  t:REV_4 input_decode[13..16]
         ELSE  REV_4    input_decode[9..12]
         ESAC       ADD_U
                 CASE sel_9_12
                 OFt: b"10110"       #add 22 to give value#
                 ELSE b"00111"       #add 7 to give value#
                 ESAC,

   mag_out_huff=CASE input_decode[1]
       OFb'0:input/0
       ELSE CASE input_decode[3]
          OFb'1:input/1
          ELSE CASE input_decode[4]
             OFb'1:input/2
             ELSE CASE input_decode[5]
                   OF     b'1:input/3
```

```
            ELSE CASE input_decode[6]
            OF b'1:input/4
            ELSE CASE input_decode[7]
              OFb'1:input/5
              ELSE CASE input_decode[8]
                OFb'1:input/6
                ELSE (S_TO_IN (b"0000" CONC mag_out2))[2]
                ESAC
              ESAC
            ESAC
          ESAC
        ESAC
      ESAC
    ESAC,

#on lpf_still bit 1 is the sign bit#
  sign = CASE mode
    OFlpf_still:input_decode[1]
    ELSE CASE mag_out_huff
      OFinput/0:b'0
      ELSE input_decode[2]
      ESAC
    ESAC,

#select huff value, 0(in lpf_send) or real value, rearange the bits for real data#
#on lpf_still bit 1 is sign bit so discard#
  mag_out = CASE mode
     OF   lpf_still:(S_TO_IN (CAST{STRING[9]bit}[INT j=1..9]input_decode[11-j]))[2]
     ELSE mag_out_huff
     ESAC,
```

```
   token_length = b"000"CONC token_length_in,

#decode token, valid only during a token cycle#
     token = CASE token_length[4..5]
       OFb"10":input_decode[1..2],
           b"01":input_decode[1] CONC b"0"
       ESAC.

JOIN   (buffer,s)     ->input_decode.

OUTPUT (sign,mag_out,token)
END.


#the huffman encoder#
FN HUFFMAN_ENCODE = (t_input:value,bit:sign,STRING[2]bit:token,t_mode:mode, t_cycle:cycle,
               STRING[16]bit:buffer,STRING[5]bit:s)
                          ->(STRING[32]bit):
BEGIN
    MAKE     SHIFT16X16_32:shift.
 #encode value#
 LET    header = CAST{STRING[2]bit}(b'1,sign),

     value_bit = CAST{[16]bit}(IN_TO_S{16} value)[2],

     sub_const = CASE value
           OF  input/(7..21):b"00111",
             input/(22..37):b"10110"
           ELSE  b"00000"
         ESAC,
```

```
  sub_value = ((IN_TO_S{input_exp}value)[2] SUB_U sub_const)[8..11],

enc_value=

 CASE cycle
 OF   token_cycle:token CONC ZERO{14}b"0",   #token is msb, max 2 bits#
   data_cycle: CASE mode
#on intra & LPF pass thro value as 16 bit word, and reverse bit order, place sign first next to lsb#
     OF   lpf_still:CAST{STRING[1]bit} sign CONC CAST{STRING[15]bit} ([INT j=1..15]value_bit[17-j])
#otherwise value is to Huffman encoded, so out 16 bit as this is the max, the shift removes the extra bits#
     ELSE   CASE value
       OF   input/0:b"0"CONC ZERO{15}b"0",
          input/1:header CONC b"1"CONC ZERO{13}b"0",
          input/2:header CONC b"01"CONC ZERO{12}b"0",
          input/3:header CONC b"001"CONC ZERO{11}b"0",
          input/4:header CONC b"0001"CONC ZERO{10}b"0",
          input/5:header CONC b"00001"CONC ZERO{9}b"0",
          input/6:header CONC b"000001"CONC ZERO{8}b"0",

          input/(7..21):header CONC b"000000" CONC(REV_4 sub_value)CONC ZERO{4}b"0",   #sub 7 to give value#

          input/(22..37):header CONC b"0000001111" CONC (REV_4 sub_value)        #sub 22 to give value#

          ELSE header CONC b"00000011111111"
          ESAC

     ESAC,

   skip_cycle:ZERO{16}b"0"    #dummy value#
  ESAC.
```

```
JOIN  (buffer ,enc_value,s[2..5])      ->shift.


#max value is 37 so 8 bits enough#
OUTPUT shift
END.


# some basic macros for the convolver, assume these will#
#be synthesised into leaf cells#
MAC  MX_4{TYPE ty}=(ty:in1 in2 in3 in4, [2]bool:sel)      ->ty:
CASE   sel
OF     (f,f):in1,
       (f,t):in2,
       (t,f):in3,
       (t,t):in4
ESAC.

MAC ENCODE4_2 = (t_mux4:in)     ->[2]bool:
CASE in
OF     uno:(f,f),
       dos:(f,t),
       tres:(t,f),
       quatro:(t,t)
ESAC.

MAC ENCODE3_2 = (t_mux3:in)     ->[2]bool:
CASE in
OF     l:(f,f),
       c:(f,t),
```

```
    r:(t,f)
ESAC.

MAC MUX_3{TYPE t}=(t:in1 in2 in3 , t_mux3:sel) ->t:
MX_4{t}(in1,in2,in3,in1,ENCODE3_2 sel).

MAC MUX_4{TYPE t}=(t:in1 in2 in3 in4, t_mux4:sel)    ->t:
MX_4{t}(in1,in2,in3,in4,ENCODE4_2 sel).


MAC MUX_2{TYPE t}=(t:in1 in2, t_mux:sel)   ->t:
CASE sel
OF    left:in1,
   right:in2
ESAC.

MAC MUX_8{TYPE ty}=(ty:in1 in2 in3 in4 in5 in6 in7 in8, [3]bool:sel) ->ty:
CASE sel
OF    (f,f,f):in1,
   (f,f,t):in2,
     (f,t,f):in3,
     (f,t,t):in4,
     (t,f,f):in5,
     (t,f,t):in6,
     (t,t,f):in7,
     (t,t,t):in8
ESAC.
MAC MX16=(STRING[16]bit:in, STRING[4]bit:sel)   ->bit:
CASE sel
OF    b"0000":in[1],
      b"0001":in[2],
```

```
    b"0010":in[3],
    b"0011":in[4],
    b"0100":in[5],
    b"0101":in[6],
    b"0110":in[7],
    b"0111":in[8],
    b"1000":in[9],
    b"1001":in[10],
    b"1010":in[11],
    b"1011":in[12],
    b"1100":in[13],
    b"1101":in[14],
    b"1110":in[15],
    b"1111":in[16]
ESAC.
COM
MAC MX16=(STRING[16]bit:in, STRING[4]bit:sel)   ->bit:
MUX_2{bit}(
    MUX_8{bit}(in[1],in[2],in[3],in[4],in[5],in[6],in[7],in[8],CAST{[3]bool}sel[2..4]),
    MUX_8{bit}(in[9],in[10],in[11],in[12],in[13],in[14],in[15],in[16],CAST{[3]bool}sel[2..4]),
    CASE sel[1]
    OF  b'0:left
    ELSE right
    ESAC).
MOC

MAC INT_BOOL = (t_quant:q)    ->[3]bool:
CASE q
OF     quant/0:(f,f,f),
       quant/1:(f,f,t),
       quant/2:(f,t,f),
```

```
    quant/3:(f,t,t),
    quant/4:(t,f,f),
    quant/5:(t,f,t),
    quant/6:(t,t,f),
    quant/7:(t,t,t)
ESAC.

COM
MAC  MUX_3{TYPE t}=(t:in1 in2 in3, t_mux3:sel) ->t:
CASE sel
OFl:in1,
   c:in2,
   r:in3
ESAC.

MAC  MUX_4{TYPE t}=(t:in1 in2 in3 in4, t_mux4:sel) ->t:
CASE sel
OFuno:in1,
   dos:in2,
   tres:in3,
   quatro:in4
ESAC.
MOC

FN NOT = (bool:in)->bool:CASE in OF t:f,f:t ESAC.

FN XOR = (bool: a b)   ->bool:
CASE (a,b)
OF  (f,f)|(t,t):f
ELSE t
ESAC.
```

```
FN AND = (bool: a b)    ->bool:
CASE (a,b)
OF  (t,t):t,
   (f,bool)|(bool,f):f
ESAC.

FN OR = (bool: a b)    ->bool:
CASE (a,b)
OF  (f,f):f,
(t,bool)|(bool,t):t
ESAC.

MAC DEL{TYPE t} = (t)     ->t:DELAY(?t,1).

#a general d latch#
MAC LATCH {TYPE t}=(t_load:load,t:in)  ->t:
BEGIN
MAKE DEL{t}:del.
LET out=CASE load
      OF   write:in
      ELSE  del
      ESAC.
JOIN out->del.
OUTPUT out
END.

#a general dff#
MAC DF1 {TYPE t}=(bool:ck,t:in)    ->t:
BEGIN
MAKE DEL{t}:del.
```

```
JOIN in->del.
OUTPUT del
END.

#a resetable DFF, init value is input parameter#
MAC DFF_INIT{TYPE t}=(bool:ck,t_reset:reset,t_load:load,t:in init_value)    ->t:
BEGIN
MAKE DEL{t}:del.
LET out=CASE (load,reset)
      OF    (write,t_reset):in,
            (read,rst):init_value
      ELSE  del
      ESAC.
JOIN out->del.
OUTPUT CASE reset
      OF rst:init_value
      ELSE   del
      ESAC
END.

#a dff resetable non- loadable dff#
MAC DFF{TYPE t}=(bool:ck,t_reset:reset,t:in init_value)    ->t:
BEGIN
MAKE DEL{t}:del.
JOIN in->del.
OUTPUT CASE reset
      OF rst:init_value
      ELSE   del
      ESAC
END.
```

```
MAC PDEL{TYPE t,INT n} = (t:in) ->t:
    IF n=0 THEN DEL{t}in
    ELSE   PDEL{t,n-1} DEL{t} in
    FI.

MAC PDF1{TYPE t,INT n} = (bool:ck,t:in ) ->t:
    IF n=0 THEN DF1{t}(ck,in)
    ELSE   PDF1{t,n-1}(ck,DF1{t}(ck, in))
    FI.
#generates the new_mode from the old, and outputs control signals to the tokeniser#

FN MODE_CONTROL = (bool:ck, t_reset:reset, t_intra:intra_inter,bool:lpf_done,[7]bool:flags,
      STRING[2]bit:token_in,t_octave:octave,t_state:state,t_direction:direction,t_load:load_mode_in
      ,t_cycle:cycle)
           ->(t_mode,t_mode,STRING[2]bit,t_diff,STRING[2]bit,t_mode):
#new_mode,proposed mode,current token,difference,token_length, #
BEGIN

   MAKE [4]DFF_INIT{t_mode}:mode,
      DFF_INIT{t_diff}:diff_out,
      DFF_INIT{t_mode}:next_mode.
 LET   nzflag=flags[1],
    origin=flags[2],
    noflag=flags[3],
    ozflag=flags[4],
    motion=flags[5],
    pro_new_z = flags[6],
    pro_no_z = flags[7],
```

```
    lpf_done_del = DFF{bool}(ck,reset,lpf_done,f).        #synchronise mode change at end of LPF#

LET next = (SEQ
#the proposed value for the mode at that octave, flags etc will change this value as necessary#
#proposed, or inherited mode from previous tree#

        VAR    pro_mode:= CASE reset
                 OF  rst:CASE intra_inter    #reset on frame start, so do lpf#
                      OF intra:lpf_still
                      ELSE lpf_send
                      ESAC
                 ELSE  CASE lpf_done_del
            OFt:CASE intra_inter    #store default mode in mode[4]#
                        OF intra:still
                        ELSE send
                        ESAC

            ELSE CASE state
               OFdown1:mode[3], #jump sideways in oct/1#
               ,  up0:mode[4]
               ELSE CASE octave
                  OF    oct/0:mode[1],
                     oct/1:mode[2],
                     oct/2:mode[3]
                  ESAC
               ESAC
            ESAC

           ESAC,
       new_mode:=pro_mode,        #inherit the previous mode#
       token_out:=b"00",
```

```
        difference:=nodiff,
        token_length:=b"00",
        flag:=f;
    CASE direction
    OF forward:

     CASE pro_mode
     OFvoid:CASE ozflag
            OF    t:new_mode:=stop
            ELSE
            ESAC,              #stay in these modes until end of tree#

       void_still: ,              #intra so must zero out all of tree#

         still_send:(token_length:=b"01";
                     CASE (nzflag OR pro_new_z)
                       OF   t:(token_out:=b"00";
                              CASE ozflag
                              OF     t:new_mode:=stop
                              ELSE   new_mode:=void
                              ESAC)
                       ELSE (token_out:=b"10";
                          new_mode:=still_send)
                       ESAC
         ),

       send: CASE ozflag
          OFt:(token_length:=b"01";

              CASE (nzflag OR pro_new_z )
```

```
   OF   t:(token_out:=b"00";
      new_mode:=stop)
   ELSE    (token_out:=b"10";
     new_mode:=still_send)
   ESAC
   )
ELSE (token_length:=b"10";

  CASE ( (NOTnoflag OR motion) AND NOTnzflag)
  OFt:( CASE origin
        OF   t:flag:=pro_new_z
        ELSE  (flag:=pro_no_z;
          difference:=diff)
        ESAC;
     CASE flag
     OFt:(token_out:=b"10";
         new_mode:=void)
     ELSE     CASE origin
            OFt:(token_out:=b"01";
                 new_mode:=still_send)
            ELSE (token_out:=b"11";
                new_mode:=send)
            ESAC
     ESAC)
  ELSE

     CASE (motion OR origin)AND nzflag
     OFt:(token_out:=b"10";
         new_mode:=void)
     ELSE (token_out:=b"00";
        new_mode:=stop)
```

```
            ESAC
          ESAC
          )
        ESAC,

  still:   (token_length:=b"01";
        CASE (nzflag OR pro_new_z)
        OFt:(token_out:=b"00";
                   new_mode:=void_still)   #zero out tree#
        ELSE (token_out:=b"10";
                  new_mode:=still)
        ESAC
        ),

  (lpf_still):(token_out:=b"00";           #for ELLA only DUMBII#
         token_length:=b"00"),

  (lpf_send):(difference:=diff;
         token_length:=b"01";

         CASE (noflag OR pro_no_z)
              OF     t:(token_out:=b"00";
              new_mode:=lpf_stop)
         ELSE (token_out:=b"10";
              new_mode:=lpf_send) #as mode stop but for this block only#
         ESAC)
  ESAC,

inverse:
  CASE pro_mode
```

```
OF void: CASE ozflag
        OF   t:new_mode:=stop
        ELSE
        ESAC,

    void_still: ,

    send:  CASE ozflag
        OFt:(token_length:=b"01";     #repeat of still-send code#
                    CASE token_in[1]
                    OF      b'1:new_mode:=still_send,
                         b'0:new_mode:=stop
                    ESAC
                    )

        ELSE (token_length:=b"10";
            CASE token_in
            OFb"11": (difference:=diff;
                    new_mode:=send),
                b"01":new_mode:=still_send,
                b"10":new_mode:=void,
                b"00":new_mode:=stop
            ESAC
            )
        ESAC,

    still_send:    (token_length:=b"01";
            CASE token_in[1]
            OFb'1:new_mode:=still_send,
                b'0:    CASE ozflag
                    OFt:new_mode:=stop
```

```
                ELSE new_mode:=void
                ESAC
            ESAC
            ),

    still:  (token_length:=b"01";
        CASE token_in[1]
                OF    b'1:new_mode:=still,
                      b'0:new_mode:=void_still
                ESAC
              ),


  (lpf_send):(difference:=diff;
          token_length:=b"01";
            CASE token_in[1]
                  OF    b'0:new_mode:=lpf_stop,
              b'1:new_mode:=lpf_send
            ESAC),

     lpf_still:
  ESAC

ESAC;

OUTPUT (new_mode,pro_mode,token_out,difference,token_length)
).

LET load_mode = CASE (reset,lpf_done_del) #store base mode in mode[3]& mode[4], base changes after lpf#
          OF    (rst,bool)|(t_reset,t):(read,read,write,write)
          ELSE  CASE (octave,load_mode_in)
```

```
                OF     (oct/1,write):(write,write,read,read),
                          (oct/2,write):(read,write,write,read)
                ELSE (read,read,read,read)
                ESAC
           ESAC.
#save the new mode& difference during a token cycle, when the flags and tokens are valid#
JOIN       (ck,reset,CASE cycle
        OF    token_cycle:write
        ELSEread
        ESAC, next[1],still)             ->next_mode,

          (ck,reset,CASE cycle
           OF    token_cycle:write
           ELSE   read
           ESAC, next[4],nodiff)                  ->diff_out.


#now write the new mode value into the mode stack at end of cycle, for later use #
FOR INT i =1..4 JOIN  (ck,no_rst,load_mode[i],CASE (reset,lpf_done_del)
                      OF(no_rst,t)|(rst,bool):next[2]
                      ELSE next_mode
                      ESAC,still)   ->mode[i].
#dont update modes at tree base from lpf data, on reset next[1] is undefined#

OUTPUT (next_mode,next[2],next[3],diff_out,next[5],next[1])
END.

#the tree coder chip#
#threshold = 2^quant_norm#

FN PALMAS= (bool:ck,t_reset:reset,t_direction:direction,t_intra:intra_inter,t_channel_factor:channel_factor,
```

```
    [4]t_quant:quant_norm, STRING[16]bit:buffer_in,
    t_input:new old,[4]t_result:threshold, t_fifo:fifo_full fifo_empty, STRING[xsize]bit:col_length,
    STRING[ysize]bit:row_length,STRING[xsize]bit:ximage_string,#ximage#
    STRING[ysize]bit:yimage_string,STRING[11]bit:yimage_string_3#yimage& yimage*2.5#)

    ->(t_input,t_sparc_addr,(t_load,t_cs),(t_load,t_cs),STRING[16]bit,[2]t_load,bool,t_cycle):

#old,address,(rw_new,cs_new),(rw_old,cs_old),buffer_out,fifo_read fifo_write, cycle#

BEGIN
    MAKEDECIDE:decide,
        ADDR_GEN:addr_gen,
        HUFFMAN_ENCODE:huffman_encode,
        FIFO_BUFFER:fifo_buffer,
        HUFFMAN_DECODE:huffman_decode,
        MODE_CONTROL:mode,
        CONTROL_COUNTER:control_counter,
        BLK_SUB_COUNT: sub_count,
        DFF_INIT{t_channel}:channel,
        QUANT:quant.

LET

    nzflag=decide[1],
    origin=decide[2],
    noflag=decide[3],
    ozflag=decide[4],
    motion=decide[5],
    pro_no_z = decide[7],#pro_no_z or pro_new_z#
    pro_new_z = decide[6],
```

```
new_mode = mode[1],
pro_mode = mode[2],
token_out = mode[3],
difference = mode[4],
token_length = mode[5],

pro    =quant[1],    #pro_no, or pro_new#
lev_out = (S_TO_IN quant[2])[2], #corresponding level#
sign = quant[3],  #and sign #

octs = addr_gen[2],
sub_en = addr_gen[3],
tree_done = addr_gen[4],
lpf_done = addr_gen[5],
state = addr_gen[6],

cycle =control_counter[2],
cs_new=control_counter[7],
rw_new=read,
rw_old=control_counter[8],
cs_old=control_counter[9],

load_channel= CASE (sub_en,sub_count[2])  #change channel#
        OF(t,t):write
        ELSE read
        ESAC,

new_channel = CASE channel_factor
            OF     luminance:y
            ELSE   CASE channel
                 OF     y:u,
```

```
                  u:v,
                  v:y
              ESAC
          ESAC,
#flush the buffer in the huffman encoder#
   flush_buffer =DFF{bool}(ck,reset,CASE channel_factor
              OFluminance:CASE load_channel
                     OF   write:t
                     ELSE    f
                     ESAC,
                 color: CASE (channel,load_channel)
                    OF(v,write):t
                    ELSE f
                    ESAC
              ESAC,f)),

   frame_done = PDF1{bool,1}(ck,flush_buffer),

   fifo_write=fifo_buffer[6],
   fifo_read =fifo_buffer[5],
   s =fifo_buffer[4],

   buffer_out = fifo_buffer[1],

     lev_in = huffman_decode[2],
     sign_in = huffman_decode[1],
     token_in = huffman_decode[3],


   del_new = PDF1{t_input,4}(ck,new),
```

```
#old has variable delays for inverse#
  del_old = CASE (direction,pro_mode)
      OF   (forward,t_mode)|(inverse,send|still_send|lpf_send|void): PDF1{t_input,4}(ck,old)
      ELSE    PDF1{t_input,1}(ck,old)
      ESAC,
  decide_reset=CASE reset
           OF rst:rst
         ELSE  control_counter[3]
        ESAC,

  oct_sel = CASE pro_mode
        OFlpf_still|lpf_send|lpf_stop:quatro
        ELSE CASE (octs,channel)
           OF (oct/0,y):uno,
              (oct/1,y)|(oct/0,u|v):dos,
              (oct/2,y)|(oct/1,u|v):tres
           ESAC
        ESAC,

  threshold_oct = MUX_4{t_result}(threshold[1],threshold[2],threshold[3],threshold[4],oct_sel),

  quant_oct = MUX_4{t_quant}(quant_norm[1],quant_norm[2],quant_norm[3],quant_norm[4],oct_sel).

JOIN  (ck,decide_reset,threshold_oct,new,old,threshold_oct,threshold_oct,octs,control_counter[6])->decide,

(ck,reset,intra_inter,lpf_done,decide,token_in,octs,state,direction,control_counter[1],cycle)->mode,

#delay the new&old values by 5 or 1 depending on mode & direction#
   ( (IN_TO_S{input_exp}del_new)[2], (IN_TO_S{input_exp}del_old)[2],
   (IN_TO_S{input_exp}lev_in)[2], sign_in,direction,quant_oct,difference,pro_mode)  ->quant,
```

```
(ck,reset ,new_channel,channel,load_channel,sub_count[1],col_length,row_length,
ximage_string,yimage_string,yimage_string_3,control_counter[4],control_counter[5],new_mode)->addr_gen,

(ck,reset,direction,cycle,pro_mode,lev_out,huffman_decode[2],buffer_in,fifo_full,
fifo_empty,huffman_encode, token_length, flush_buffer,quant_norm[4])            ->fifo_buffer,

(lev_out ,sign,token_out,pro_mode,cycle,fifo_buffer[2],s)            ->huffman_encode,

  (pro_mode,token_length,fifo_buffer[2] CONC fifo_buffer[3],fifo_buffer[4])      ->huffman_decode,

(ck,reset,sub_en,t,t)                      ->sub_count,

(ck,reset,pro_mode,new_mode,direction)               ->control_counter,

(ck,reset,load_channel,new_channel,y)              ->channel.
OUTPUT

 ( CASE new_mode
  OF   void|void_still:input/0
  ELSE    (S_TO_INpro)[2]
  ESAC            ,addr_gen[1],(rw_new,cs_new),(rw_old,cs_old),buffer_out,(fifo_read,fifo_write),frame_done,cycle)
END.
COM
#the decoder for the barrel shifter-- decides if the bit value and q value are  #
#in the upper-triangle, or diagonal and set the control bits            #

MAC DECODE{INT n} = (t_quant:q)     ->[qmax](bool#upper diag#,bool#diagonal#):
BEGIN
   #one bit of the decoder#
   MAC  DECODE_BIT{INT j}= (t_quant:q)    ->(bool,bool):
   CASE q
```

```
    OF   quant/(0..qmax-j):(t,f),     #upper triangle#
        quant/(qmax-j+1):(f,t)   #diagonal#
    ELSE  (f,f)
    ESAC.
OUTPUT([INT j=1..qmax]DECODE_BIT{j}(q))
END.

#now the selector fn to mux between the data_in bit ,0 or 1 depending on q#
MAC SELECTOR = (t_quant:q,STRING[INT n]bit:data)
                  ->(STRING[n]bit#level#,STRING[n]bit#round_level#):
BEGIN
    #the 3->2 bit selector#
    MAC SELECT_BIT = ([2]bool:upper_or_diag,bit:data)      ->(bit,bit):#level[j],round_level[j]#
    CASE upper_or_diag
    OF      (t,f):(data,data),       #upper-triangle#
            (f,t):(b'0,b'0)         #diagonal#
    ELSE    (b'0,b'1)              #lower-triangle#
    ESAC.
 MAKE    DECODE{n}:decode,
       [qmax]SELECT_BIT: select.
 JOIN    (q)     ->decode.

 FOR INT j=1..qmax JOIN (decode[j],data[n-qmax+j])      ->select[j].

 OUTPUT (data[1..n-qmax] CONC (BIT_STRING{qmax}([INT j=1..(qmax)]select[j][1]) ), #level#
       data[1..n-qmax] CONC (BIT_STRING{qmax}([INT j=1..(qmax)]select[j][2]) ) #round_level#
       )
 END.
 MOC

 #now the selector fn to shift the level depending on q#
```

```
MAC BARREL_SHIFT_RIGHT = (t_quant:q,STRING[INT n]bit:data) ->(STRING[n]bit#level#):
MUX_8{STRING[n]bit}(
     data,
     b"0"CONC data[1..n-1],
     b"00"CONC data[1..n-2],
     b"000"CONC data[1..n-3],
     b"0000"CONC data[1..n-4],
     b"00000"CONC data[1..n-5],
     b"000000"CONC data[1..n-6],
     b"0000000"CONC data[1..n-7],
     INT_BOOL q).

#the bshift for the inverse, to generate the rounded level #
MAC BARREL_SHIFT_LEFT = (t_quant:q,STRING[INT n]bit:data#lev#) ->(STRING[n]bit#round_level#):
MUX_8{STRING[n]bit}(
     data,
     data[2..n]CONCb"0",
     data[3..n]CONCb"01",
     data[4..n]CONCb"011",
     data[5..n]CONCb"0111",
     data[6..n]CONCb"01111",
     data[7..n]CONCb"011111",
     data[8..n]CONCb"0111111",
     INT_BOOL q).

#the function to return the quantised level(UNSIGNED), and proposed value given,#
# the new&old values, forw/inverse direction  #

FN QUANT = (STRING[input_exp]bit: new old lev_inv,bit:sign_lev_inv, t_direction:direction,t_quant:q,t_diff:difference,
       t_mode:mode)
```

```
        -> (STRING[input_exp]bit,STRING[input_exp]bit,bit) #pro,lev& sign# :

BEGIN

LET
#decide which of new-old or new will be quantised, and the sign of the level#
#level is stored in sign &magnitude form#

   dir_sel =     CASE direction
                 OF    forward:left,
                       inverse:right
                 ESAC,

   sub_sel = CASE difference
                OF    diff:left
                ELSE  right      #put old=0#
                ESAC,

  sub_in= MUX_2{STRING[input_exp]bit}(old,ZERO{input_exp}b"0",sub_sel),


  no =ADD_SUB_ST(new,sub_in,subt),

  lev_final= ABS_S no,      #now input_exp+1 bits#

   sgn_level = MUX_2{bit}( #sign of value to be quantised#
                 no[1],
                 sign_lev_inv,
                 dir_sel).
```

```
#find the quant. level by shifting by q, for the inverse it comes from the Huffman decoder#

LET
   lev_data = BARREL_SHIFT_RIGHT(q,lev_final),

#saturate the lev at 37, for the Huffman table,except in lpf_still mode, send all the bits#
   lev_forw = CASE mode
        OF  lpf_still:lev_data
        ELSE   CASE lev_data GT_U b"00000100101"
          OFt:b"00000100101"
          ELSE lev_data
          ESAC
        ESAC,

    lev  = MUX_2{STRING[input_exp+1]bit}(
                 lev_forw,
                 b"0" CONC lev_inv,
            dir_sel),
 #the level = 0 flag#
      lev_z = lev EQ_U ZERO{input_exp+1}b"0",

    inv_lev_z =  CASE lev_z
               OF t:b'0
                 ELSE b'1
            ESAC,

 #the level value shifted up, and rounded#
      round_lev = BARREL_SHIFT_LEFT(q,lev) AND_B
           CASE mode
            OF  lpf_still:b"00" CONC ALL_SAME{input_exp-1}b"1"
            ELSE        BIT_STRING{input_exp+1}([input_exp+1]inv_lev_z) #if lev==0 out all 0's#
```

```
            ESAC,
#clear out extra bit for lpf_still case#


#calculate the proposed value:in the case n-o,round_lev is unsigned 10 bit, so result needs 11 bits#
#pro_no will always be in range as round_lev<|n-o|        #

   pro_no = ADD_SUB_ST(old,round_lev,CASE sgn_level
                OFb'0:add,
                   b'1:subt
                ESAC),

#now pro_new = +/- round_lev#

   round_sel = CASE sgn_level
          OFb'0: left,
             b'1: right
          ESAC,

     pro_new = MUX_2{STRING[input_exp+1]bit}(
          round_lev ,
          (NEG_U round_lev)[2..input_exp+2],     #NEG sign extends#
       round_sel),

    out_sel = CASE difference
       OFdiff:left
       ELSE right
       ESAC.


  OUTPUT (MUX_2{STRING[input_exp]bit}(
```

```
                pro_no[3..input_exp+2],
                pro_new[2..input_exp+1],
                out_sel) ,
        lev[2..input_exp+1] ,
        sgn_level)

END.
#actel 1 bit full adder with active low cin and cout#
FN FA1B = (bit: ain bin cinb)     ->(bit,bit):#cob,s#
BEGIN
LET     a_c =B_TO_S ain CONC NOT_B(B_TO_S cinb) ,
        b_c = B_TO_S bin CONC NOT_B(B_TO_S cinb) ,
      out = ADD_U(a_c,b_c).
OUTPUT(CAST{bit} NOT_B(B_TO_S out[1]), out[2])
END.


#a Ripple carry adder using 1 bit full adder blocks#

#the actel version of the ADD BIOP's#

MAC ADD_S_ACTEL = (STRING[INT m]bit:ain,STRING[INT n]bit:bin,bit:cinb) ->STRING[IF m>=n THEN m+1 ELSE n+1 FI]bit:
BEGIN
     MAKE [IF m>=n THEN m ELSE n FI]FA1B:sum.

#signed nos so sign extend #
 LET a_c = IF m>=n THEN ain ELSE ALL_SAME{n-m}B_TO_S ain[1]  CONC ain FI,
       b_c = IF n>=m THEN bin ELSE ALL_SAME{m-n}B_TO_S bin[1] CONC bin  FI.
     LET subsignal = sum.
#lsb#
```

```
JOIN    (a_c[IF m>=n THEN m ELSE n FI],b_c[IF m>=n THEN m ELSE n FI],cinb)    ->sum[IF m>=n THEN m ELSE n FI].

FOR INT j=1..(IF m>=n THEN m ELSE n FI) -1
JOIN (a_c[(IF m>=n THEN m ELSE n FI) -j],b_c[(IF m>=n THEN m ELSE n FI) -j],
     sum[(IF m>=n THEN m ELSE n FI) -j+1][1])             ->sum[(IF m>=n THEN m ELSE n FI) -j].

OUTPUT   CAST{STRING[IF m>=n THEN m+1 ELSE n+1 FI]bit}
(NOT_B( B_TO_S sum[1][1]) CONC
            CAST{STRING[IF m>=n THEN m ELSE n FI]bit}( [INT j=1..IF m>=n THEN m ELSE n FI] sum[j][2]))
END.


MAC ADD_US_ACTEL = (STRING[INT m]bit:ain,STRING[INT n]bit:bin,bit:cinb) ->STRING[IF m>=n THEN m+1 ELSE n+1 FI]bit:
BEGIN
     MAKE [IF m>=n THEN m ELSE n FI]FA1B:sum.

#unsigned nos so extend by 0#
     LET a_c = IF m>=n THEN ain ELSE ZERO{n-m}b"0" CONC ain FI,
         b_c = IF n>=m THEN bin ELSE  ZERO{m-n}b"0" CONC bin FI.
     LET subsignal = sum.

#lsb#
JOIN    (a_c[IF m>=n THEN m ELSE n FI],b_c[IF m>=n THEN m ELSE n FI],cinb)    ->sum[IF m>=n THEN m ELSE n FI].

FOR INT j=1..(IF m>=n THEN m ELSE n FI) -1
JOIN (a_c[(IF m>=n THEN m ELSE n FI) -j],b_c[(IF m>=n THEN m ELSE n FI) -j],
     sum[(IF m>=n THEN m ELSE n FI) -j+1][1])             ->sum[(IF m>=n THEN m ELSE n FI) -j].

OUTPUT   CAST{STRING[IF m>=n THEN m+1 ELSE n+1 FI]bit}
(NOT_B(B_TO_S sum[1][1]) CONC
            CAST{STRING[IF m>=n THEN m ELSE n FI]bit}([INT j=1..IF m>=n THEN m ELSE n FI] sum[j][2]) )
```

```
END.

MAC ADD_SUB_ST =(STRING[INT m]bit:ain,STRING[INT n]bit:bin,t_add:sel) ->STRING[IF m>=n THEN m+1 ELSE n+1 FI]bit:

BEGIN

#sign extend inputs#
     LET a_s = CAST{STRING[1]bit}ain[1] CONC ain,
       b_s = CAST{STRING[1]bit}bin[1] CONC bin,
       sel_bit = CAST{STRING[1]bit}sel,
#ACTEL#
     bin_inv = XOR_B{n+1}(b_s, ALL_SAME{n+1}sel_bit),

#cinb is active low so cast sel(add->0,sub->1) & invert it#
       out= ADD_S_ACTEL(a_s,bin_inv,CAST{bit}NOT_B sel_bit),
     binout= out[2..IF m>=n THEN m+2 ELSE n+2 FI]

OUTPUT binout
END.

#transformation ops#
MAC B_TO_S= (bit:in)    ->STRING[1]bit: CASE in
                          OF b'0:b"0",
                            b'1:b"1"
                   ESAC.
MAC I_TO_SC{INT n} = (t_result: in)      -> (flag,STRING[n]bit): BIOP TRANSFORM_S.
MAC SC_TO_I{INT n} = (STRING[n]bit:in)   -> (flag,t_result): BIOP TRANSFORM_S.

MAC S_TO_IN = (STRING[INT n]bit:in) -> (flag,t_input): BIOP TRANSFORM_S.
MAC IN_TO_S{INT n} = (t_input: in)     -> (flag,STRING[n]bit): BIOP TRANSFORM_S.
```

```
MAC U_TO_IN = (STRING[INT n]bit:in) -> (flag,t_input): BIOP TRANSFORM_US.

MAC U_TO_LEN = (STRING[INT n]bit:in) -> (flag,t_length): BIOP TRANSFORM_US.
MAC LEN_TO_U{INT n} =(t_length:in)  -> (flag,STRING[n]bit): BIOP TRANSFORM_US.

MAC Q_TO_U{INT n} =(t_quant:in)      -> (flag,STRING[n]bit): BIOP TRANSFORM_US.
MAC S_TO_C  = (STRING[INT n]bit:in)    ->(flag,t_col):BIOP TRANSFORM_US.
MAC S_TO_R  = (STRING[INT n]bit:in)    ->(flag,t_row):BIOP TRANSFORM_US.
MAC S_TO_B  = (STRING[INT n]bit:in)    ->(flag,t_blk):BIOP TRANSFORM_US.
MAC S_TO_SUB  = (STRING[INT n]bit:in)    ->(flag,t_sub):BIOP TRANSFORM_US.
MAC S_TO_SPARC  = (STRING[INT n]bit:in)    ->(flag,t_sparc_addr):BIOP TRANSFORM_US.

MAC C_TO_S{INT n} = (t_col: in)    -> (flag,STRING[n]bit): BIOP TRANSFORM_US.
MAC R_TO_S{INT n} = (t_row: in)    -> (flag,STRING[n]bit): BIOP TRANSFORM_US.

MAC I_TO_Q = (t_input:in)    ->t_quant:ARITH in.


MAC B_TO_I= (bit:in)    ->t_result: CASE in
                          OF b'0:result/0,
                            b'1:result/1
                      ESAC.

MAC CARRY= (t_add:in)   ->STRING[1]bit: CASE in
                          OF add:b"0",
                            subt:b"1"
                      ESAC.
MAC BOOL_BIT = (bool:in)      ->STRING[1] bit:
CASE in
OF  t:b"1"
```

```
ELSE b"0"
ESAC.

MAC BOOL_STRING{INT n} = ([n]bool:in)   ->STRING[n] bit:
(LET out = BOOL_BIT in[1].
OUTPUT IF n=1
      THEN out
      ELSE out[1] CONC  BOOL_STRING{n-1}(in[2..n])
      FI
).

MAC BIT_STRING{INT n} = ([n]bit:in)   ->STRING[n] bit:
(LET out = B_TO_S in[1].
OUTPUT IF n=1
      THEN out
      ELSE out[1] CONC  BIT_STRING{n-1}(in[2..n])
      FI
).

MAC ZERO{INT n} = (STRING[1]bit:dummy)  ->STRING[n]bit:
IF n=1 THEN b"0"
ELSE b"0" CONC ZERO{n-1} b"0"
FI.

MAC ALL_SAME{INT n} = (STRING[1]bit:dummy)  ->STRING[n]bit:
IF n=1 THEN dummy
ELSE dummy CONC ALL_SAME{n-1} dummy
FI.
```

```
COM
The operators described in this section are optimal and take two-valued
operands and produce a two-valued result.  They may not be used with
ELLA-integers or associated types.

The first basic value of any two-valued type declaration of the operand(s) and
the result are interpreted by the operations as false, and the second basic
value is interpreted as true.  Thus, given the following type declarations:

MOC

MAC AND_T = (TYPE t: a b ) -> t: BIOP AND.

MAC OR_T = (TYPE t: a b ) -> t: BIOP OR.

MAC XOR_T = (TYPE t: a b ) -> t: BIOP XOR.

MAC NOT_T = (TYPE t: a ) -> t: BIOP NOT.

COM
The following operations take bit-string operand(s) and are bitwise, ie the
operation is performed on the operand(s) one bit at a time.  The operand(s)
and result must all be ELLA-strings of the same length.
MOC

MAC AND_B = (STRING[INT n]bit,STRING[n]bit) -> STRING[n]bit:
BIOP AND.

MAC OR_B = (STRING[INT n]bit,STRING[n]bit) -> STRING[n]bit:
BIOP OR.
```

```
MAC XOR_B = (STRING[INT n]bit,STRING[n]bit) -> STRING[n]bit:
BIOP XOR.

MAC NOT_B = (STRING[INT n]bit) -> STRING[n]bit:
BIOP NOT.


COM
The operators described in this section may be used with primitive types ie
all enumerated types, except associated types, rows, strings and structures.
These operations take two operands which must be of the same type and the
result can be any two-valued type; we have packaged these BIOPs so they output
a value of type 'bool' - you may change this if you wish.
MOC

MAC EQ = (TYPE t: a b ) -> bool: BIOP EQ.

MAC GT = (TYPE t: a b ) -> bool: BIOP GT.

MAC GE = (TYPE t: a b ) -> bool: BIOP GE.

MAC LT = (TYPE t: a b ) -> bool: BIOP LT.

MAC LE = (TYPE t: a b ) -> bool: BIOP LE.

COM
NOTE: these BIOPs are designed to take any primitive ELLA type. Since it is
not possible to distinguish between primitive and other types, whilst leaving
the macro declaration general enough to allow the use of all two-valued types
that might be declared, there are type-checking limitations. This is done
at network assembly, so use of illegal types will not generate an error
```

```
message until then.

NB: ARITH provides for relational operations on ELLA-integer types.
MOC

COM
These operations are optimal in their handling of '?' and operate on
bit-string representations of unsigned integers. The result may be any
two-valued type; we have used type 'bool'. The inputs can be of different
lengths and different types.
MOC

MAC EQ_U = (STRING[INT n]bit,STRING[INT m]bit) -> bool:
BIOP EQ_US.

MAC GT_U = (STRING[INT n]bit,STRING[INT m]bit) -> bool:
BIOP GT_US.

MAC GE_U = (STRING[INT n]bit,STRING[INT m]bit) -> bool:
BIOP GE_US.

MAC LT_U = (STRING[INT n]bit,STRING[INT m]bit) -> bool:
BIOP LT_US.

MAC LE_U = (STRING[INT n]bit,STRING[INT m]bit) -> bool:
BIOP LE_US.

# Bit-strings representing signed numbers #
COM
These operations are optimal and operate on bit-string representations of
signed integers. The result may be any two-valued type; we have used type
```

'bool'. The inputs can be of different lengths and different types.
MOC

MAC EQ_S = (STRING[INT n]bit,STRING[INT m]bit) -> bool:
BIOP EQ_S.

MAC GT_S = (STRING[INT n]bit,STRING[INT m]bit) -> bool:
BIOP GT_S.

MAC GE_S = (STRING[INT n]bit,STRING[INT m]bit) -> bool:
BIOP GE_S.

MAC LT_S = (STRING[INT n]bit,STRING[INT m]bit) -> bool:
BIOP LT_S.

MAC LE_S = (STRING[INT n]bit,STRING[INT m]bit) -> bool:
BIOP LE_S.

Shift operations
COM
These operate on bit-strings. Both the enclosing macro and the BIOP are parameterised by the number of bits to be shifted (INT p). The macro and BIOP parameters must match. Note that no bits are lost in these shift operations, so you may need to trim the result to achieve the desired effect.

SR means shift right; SL means shift left.

The macros with the suffix '_S' perform arithmetic shifts; those with the

```
suffix '_U' perform bool shifts.
MOC

MAC SL_S{INT p} = (STRING[INT n]bit) -> STRING[n + p]bit:
BIOP SL{p}.

MAC SL_U{INT p} = (STRING[INT n]bit) -> STRING[n + p]bit:
BIOP SL{p}.

MAC SR_S{INT p} = (STRING[INT n]bit) -> STRING[n + p]bit:
BIOP SR_S{p}.

MAC SR_U{INT p} = (STRING[INT n]bit) -> STRING[n + p]bit:
BIOP SR_US{p}.


# Arithmetic operations #

# Bit-strings representing unsigned numbers #

# addition. #

MAC ADD_U = (STRING[INT m]bit,STRING[INT n]bit)
              -> STRING[IF m >= n THEN m+1 ELSE n+1 FI]bit:
BIOP PLUS_US.

# subtraction on bit-string representations of unsigned integers.  Output is #
# signed. #

MAC SUB_U = (STRING[INT m]bit,STRING[INT n]bit)
              -> STRING[IF m >= n THEN m+1 ELSE n+1 FI]bit:
```

```
BIOP MINUS_US.

# negation.  Output is signed. #

MAC NEG_U = (STRING[INT n]bit) -> STRING[n+1]bit:
BIOP NEGATE_US.

# multiplication. #

MAC MULT_U = (STRING[INT m]bit,STRING[INT n]bit) -> STRING[m+n]bit:
BIOP TIMES_US.

COM
- divide.  If the divisor is non-zero then the first element of the output is
'ok' and the second and third elements are the quotient and remainder;
otherwise, the first element is 'error' and the rest is set to '?'.
MOC

MAC DIV_U = (STRING[INT m]bit,STRING[INT n]bit)
               -> (flag,STRING[m]bit,STRING[n]bit):
BIOP DIVIDE_US.

# square root. #

MAC SQRT_U = (STRING[INT n]bit) -> STRING[(n+1) % 2]bit:
BIOP SQRT_US.
COM
 modulus (result always positive).  If the divisor is non-zero, then the first
element of the output is 'ok' and the second element is the modulus;
otherwise, the first element is 'error'  and the second is '?'.
```

```
MOC

MAC MOD_U = (STRING[INT m]bit,STRING[INT n]bit)
                    -> (flag,STRING[n]bit):
BIOP MOD_US.

COM
- convert between one range of bit-string and another.  If the input value
cannot be represented as a legal value for the output string, the result is
'error' and '?'.
MOC

MAC RANGE_U {INT m} = (STRING[INT n]bit)
                        -> (flag,STRING[m]bit):
BIOP RANGE_US.

# Bit-strings representing signed numbers #

# addition. #

MAC ADD_S = (STRING[INT m]bit,STRING[INT n]bit)
                -> STRING[IF m >= n THEN m+1 ELSE n+1 FI]bit:
BIOP PLUS_S.

# subtraction. #

MAC SUB_S = (STRING[INT m]bit,STRING[INT n]bit)
                -> STRING[IF m >= n THEN m+1 ELSE n+1 FI]bit:
BIOP MINUS_S.

# negation. #
```

```
MAC NEG_S = (STRING[INT n]bit) -> STRING[n+1]bit:
BIOP NEGATE_S.

# multiplication. #

MAC MULT_S = (STRING[INT m]bit,STRING[INT n]bit) -> STRING[m+n]bit:
BIOP TIMES_S.

COM
divide. If the divisor is non-zero then the first element of the output is
'ok 'and the second and third elements are the quotient and remainder;
otherwise, the first element is 'error' and the rest is set to '?'. The
remainder has the same sign as the divisor.
MOC

MAC DIV_S = (STRING[INT m]bit,STRING[INT n]bit)
          -> (flag,STRING[m]bit,STRING[n]bit):
BIOP DIVIDE_S.

COM
modulus (result always positive). If the divisor is non-zero, then the first
element of the output is 'ok' and the second element is the unsigned modulus;
otherwise, the first element is 'error' and the second is '?'.
MOC

MAC MOD_S = (STRING[INT m]bit,STRING[INT n]bit)
          -> (flag,STRING[n]bit):
BIOP MOD_S.

COM
```

```
- convert between one range of bit-string and another.  If the input value
cannot be represented as a legal value for the output  string, the result is
'error' and '?'.
MOC

MAC RANGE_S {INT m}= (STRING[INT n]bit)
                    -> (flag,STRING[m]bit):
BIOP RANGE_S.

# absolute value.  The output represents an unsigned integer. #

MAC ABS_S = (STRING[INT n]bit) -> STRING[n]bit:
BIOP ABS_S.

# Built in Register #

MAC DREG{INT interval delay} = (TYPE t) -> t:
ALIEN REGISTER {interval, ?t, 0, delay}.

MAC GEN_DREG{INT interval, CONST (TYPE t): init, INT skew delay} = (t) -> t:
ALIEN REGISTER {interval, init, skew, delay}.

# Built in type conversion #

MAC CAST{TYPE t} = (TYPE s) -> t:
ALIEN CAST.
```

```
MAC ALL_SAME{INT n} = (STRING[1]bit:dummy)  ->STRING[n]bit:
BEGIN
FAULT IF n < 1 THEN "N<1 in ALL_SAME" FI.
OUTPUT IF n=1 THEN dummy
ELSE dummy CONC ALL_SAME{n-1} dummy
FI
END.

MAC CAST {TYPE to} = (TYPE from:in)     ->to:ALIEN CAST.

MAC ZERO{INT n} = (STRING[1]bit:dummy)  ->STRING[n]bit:
BEGIN
FAULT IF n < 1 THEN "N<1 in ZERO" FI.
OUTPUT IF n=1 THEN b"0"
    ELSE b"0" CONC ZERO{n-1} b"0"
    FI
END.

MAC B_TO_S= (bit:in)    ->STRING[1]bit: CASE in
                         OF b'0:b"0",
                          b'1:b"1"
                      ESAC.

MAC S_TO_IN = (STRING[input_exp]bit:in)    -> (flag,t_input): BIOP TRANSFORM_S.
MAC IN_TO_S{INT n} = (t_input: in) -> (flag,STRING[n]bit): BIOP TRANSFORM_S.

MAC S_HUFF = (STRING[6]bit)     ->(flag,t_huffman):BIOP TRANSFORM_US.
MAC HUFF_S = (t_huffman)              ->(flag,STRING[6]bit):BIOP TRANSFORM_US.


MAC BOOL_BIT = (bool:in)    ->STRING[1] bit:
```

```
CASE in
OF  t:b"1"
ELSE b"0"
ESAC.
MAC BIT_BOOL= (bit:in)       ->bool:
CASE in
OF  b'1:t
ELSE f
ESAC.


MAC BOOL_STRING{INT n} = ([n]bool:in) ->STRING[n] bit:
(LET out = BOOL_BIT in[1].
OUTPUT IF n=1
   THEN out
   ELSE out[1] CONC  BOOL_STRING{n-1}(in[2..n])
   FI
).
# defines the types used for the 2D wavelet chip#

#constant values#
INT    result_exp=14,      #length of result arith#
   input_exp=10,      #length of 1D convolver input/output#
   qmax = 7,        #maximum shift value for quantisation constant#
   result_range = 1 SL (result_exp-1),
   input_range = 1 SL (input_exp-1),
   max_octave=3, #no of octaves=max_octave +1, can not be less in this example#
   no_octave=max_octave+1,   #"#
   xsize = 10,   #no of bits for ximage#
   ysize = 9,#no of bits for yimage#
   ximage=319,#the xdimension -1 of the image, ie no of cols#
```

```
yimage=239 #the ydimension -1 of the image, ie no of rows#

#int types#
TYPE t_result= NEW result/( -(result_range)..(result_range-1)),
    t_input= NEW input/( -(input_range)..(input_range-1)),
    t_length= NEW len/(0..15),
    t_inp  = NEW inp/(0..1023),
    t_blk   =NEW blk/(0..3),
    t_sub  =NEW sub/(0..3),
    t_col   =NEW col/(0..ximage),
    t_row  =NEW row/(0..yimage),
    t_carry     =NEW carry/(0..1),
    t_quant    =NEW quant/(0..qmax),
#address for result&dwt memory, ie 1 frame#
    t_sparc_addr  =NEW addr/(0..(1 SL max_octave)*( (ximage+1)*(yimage+1)+(ximage+1))-1 ),
    t_octave=NEW oct/(0..(max_octave+1)),

#bit string and boolean types types#
    bit = NEW b('0 | '1),
    bool = NEW (f|t),
    flag = NEW(error | ok),

#control signals#
    t_reset = NEW(rst|no_rst),
    t_load  = NEW(write|read),    #r/wbar  control#
    t_cs =   NEW(no_select|select),  #chip select control#
    t_updown= NEW(down|up),        #up/down counter control#
    t_diff=   NEW(diff|nodiff),   #diff or not in quantiser#
    t_intra = NEW(intra|inter),
```

```
#convolver mux & and types#
    t_mux = NEW(left|right),
    t_mux3 = NEW(l|c|r),
    t_mux4 = NEW(uno|dos|tres|quatro),
    t_add = NEW(add|subt),
    t_direction=NEW(forward|inverse),

#counter types#
    t_count_control=NEW(count_rst|count_carry),
    t_count_2 = NEW(one|two),
#state types#
    t_token = NEW (t_0|t_1|t_11|t_100|t_101),
    t_mode= NEW(void|void_still|stop|send|still|still_send|lpf_send|lpf_still|lpf_stop),
    t_cycle = NEW(token_cycle|data_cycle|skip_cycle),
    t_state= NEW(start|up0|up1|zz0|zz1|zz2|zz3|down1),
    t_decode = NEW(load_low|load_high),
    t_high_low = NEW(low|high),
    t_huffman  = NEW(pass|huffman),
    t_fifo = NEW(ok_fifo|error_fifo),
#types for the octave control unit#
    t_channel= NEW(y|u|v),
    t_channel_factor= NEW(luminance|color),
#types for the control of memory ports#
    t_sparcport=(t_sparc_addr#wr_addr#,t_sparc_addr#rd_addr#,t_load#w/r#,t_cs#cs#)

#generate random values for test memories#

FN GEN_RANDOM_MEM = (bool:ck,t_reset:reset)  ->t_input: BOOL_INT10 PRBS11(ck,reset).
TYPE t_test = NEW(no|yes).
#..............................................................#
#These functions change types from boolean to inputeger and vice- #
```

```
#versa. Supports 1 & 8 bit booleans.                    #
#.......................................................#

FN INT_BOOL1=(t_input:k)   ->bool:        # 1bit input to binary #
        CASE     k
        OFinput/0:f,
           input/1:t
     ESAC.

FN BOOL_INT=(bool:b)    ->t_input:     # 1 bit bool to input #
        CASE     b
        OFf:input/0,
           t:input/1
     ESAC.

FN * =(t_input:a b)  ->t_input: ARITH a*b.
FN % =(t_input:a b)->t_input: ARITH a%b.
FN - =(t_input:a b)  ->t_input: ARITH a-b.
FN + =(t_input:a b)  ->t_input: ARITH a+b.
FN = =(t_input:a b)  ->t_test: ARITH IF a=b THEN 2 ELSE 1 FI.

COM
FN CHANGE_SIGN = (t_input:i)  ->t_input: #changes sign for 8-bit 2's#
    ARITH  IF i<0 THEN 128+i          #complement no,  #
                ELSE i
                FI.

FN SIGN = (t_input:i)   ->bool:          #gets sign for 2's#
    ARITH  IF i<0 THEN 1          #complement nos   #
                ELSE 2
```

```
        FI.

FN TEST_SIZE = (t_input:x)  ->bool:
#tests to see if the input is bigger than an 8-bit inputeger#
   ARITH IF ( (x<=-128) AND (x>127)) THEN 1
                 ELSE 2     FI.


FN INT8_BOOL=(t_input:orig)    ->[8]bool:
BEGIN

   SEQ
      VAR i1:=input/0,     #input variables#
         i0:=CHANGE_SIGN(orig),
         b:=(f,f,f,f,f,f,f,SIGN(orig));
    [INT n=1..7]  (
          i1:=i0%input/2;
          b[n]:=INT_BOOL1(i0-input/2*i1);
          i0:=i1
          );
     OUTPUT CASE TEST_SIZE orig          #checks to see if orig will#
        OFt:  [8]?bool,  #fit inputo an 8_bit value#
           f:  b
        ESAC
   END.

  FN BOOL_INT8=([8]bool:b)   ->t_input: #converts 8bit boolean to 2's#
  BEGIN
     SEQ              #complement inputeger   #
        VAR  sum:=input/-128 * BOOL_INT(b[8]),
            exp:=input/1;
```

```
[INT k=1..7]
    (   sum:=sum+exp*BOOL_INT(b[k]);
        exp:=input/2 * exp
    );
  OUTPUT sum
END.


MOC
FN BOOL_INT10=([10]bool:b)    ->t_input: #converts 10bit boolean to 2's#
BEGIN
   SEQ              #complement integer  #
      VAR  sum:=input/-512 * BOOL_INT(b[10]),
          exp:=input/1;
      [INT k=1..9]
         (   sum:=sum+exp*BOOL_INT(b[k]);
             exp:=input/2 * exp
         );
     OUTPUT sum
  END.
  COM
  FN BOOL_INT16 =([8]bool:in1 in2)  ->t_input:
  # convetrs a 16-bit no., (lsbs,msbs) inputo inputeger form)#
  (BOOL_INT8(in1))+((input/256)*BOOL_INT8(in2))+((input/256)*BOOL_INT(in1[8])).
                              #hack because of sign extend#
                                 #of lsb #

  MOC
  COM
  FN PRBS10 = (t_reset:reset) ->[10]bool:
  #A 10 bit prbs generator,feedback taps on regs 3 & 10.#
  BEGIN
```

```
    MAKE[10]MYLATCH:l,
        XNOR:xnor.

    FOR INT k=1..9 JOIN
        (reset,l[k])     ->l[k+1].

    JOIN  (reset,xnor)  ->l[1],
        (l[10],l[3]) ->xnor.

    OUTPUT l
END.
MOC
FN PRBS11 = (bool:ck,t_reset:reset)    ->[10]bool:
#A 11 bit prbs generator,feedback taps on regs 2 & 11.#
BEGIN
    MAKE[11]DFF{bool}:l,
        XOR:xor.

    FOR INT k=1..10    JOIN
        (ck,reset,l[k],f)    ->l[k+1].

    JOIN  (ck,reset,NOTxor,f) ->l[1],
        (l[11],l[2]) ->xor.

    OUTPUT l[1..10]
END.
COM
FN PRBS16 = (bool:reset)->[16]bool:
#A 16 bit prbs generator,feedback taps on regs 1,3,12,16#
BEGIN
    MAKE[16]MYLATCH:l,
```

```
        XOR_4:xor,
        NOT:xnor.

    FOR INT k=1..15    JOIN
        (ck,reset,l[k]) ->l[k+1].

    JOIN  (ck,reset,xnor)   ->l[1],
        (l[1],l[3],l[16],l[12])   ->xor,
        xor     ->xnor.
    OUTPUT ([INT k=1..16]l[k])
END.
FN PRBS12 = (clock:ck,bool:reset)  ->[12]bool:
#A 12 bit prbs generator,feedback taps on regs 1,4,6,12.#
BEGIN
    MAKE[12]MYLATCH:l,
        XOR_4:xor,
        NOT:xnor.

    FOR INT k=1..11    JOIN
        (ck,reset,l[k])->l[k+1].

    JOIN  (ck,reset,xnor)   ->l[1],
        (l[1],l[4],l[6],l[12])     ->xor,
        xor     ->xnor.
    OUTPUT ([INT k=1..12]l[k])
END.

FN PRBS8 = (clock:ck,bool:reset)    ->[8]bool:
#A 8 bit prbs generator,feedback taps on regs 2,3,4,8.#
BEGIN
    MAKE[8]MYLATCH:l,
```

```
      XOR_4:xor,
      NOT:xnor.

   FOR INT k=1..7 JOIN
      (ck,reset,l[k]) ->l[k+1].

   JOIN  (ck,reset,xnor)   ->l[1],
       (l[2],l[3],l[4],l[8])  ->xor,
       xor     ->xnor.
   OUTPUT ([INT k=1..8]l[k])
   END.
MOC
#test for palmas chip#
TYPE t_int32 = NEW int32/(-2147483000..2147483000).

FN RMS = (bool:ck,t_reset:reset,t_cycle:cycle, t_input:old new)    ->t_int32:
BEGIN

FN I_32 = (t_input:in)  ->t_int32:ARITH in.
FN DV = (t_int32:a b)   ->t_int32:ARITH a%b.
FN PL = (t_int32:a b)   ->t_int32:ARITH a+b.
FN MI = (t_int32:a b)   ->t_int32:ARITH a-b.
FN TI = (t_int32:a b)   ->t_int32:ARITH a*b.

MAKEDFF_INIT{t_int32}:old_error.

LET  err = I_32old MI I_32new,
     err2 = (errTIerr) PL  old_error.

JOIN  (ck,reset,CASE cycle
        OFdata_cycle:write
```

```
        ELSE read
        ESAC,err2,int32/0)      ->old_error.

    OUTPUT old_error
    END.

FN EQ = (t_input:a b)  ->bool:ARITH IF a=b THEN 2
                ELSE 1
                FI.

FN SPARC_MEM = (t_input:in,t_sparc_addr:wr_addr,t_sparc_addr:rd_addr,t_load:rw_sparc#,t_cs:cs#)->t_input:
        RAM(input/0).

FN FIFO     =(bool:ck,t_reset:reset,STRING[16]bit:buffer_in,t_direction:direction,t_load:fifo_read fifo_write)
                ->(STRING[16]bit,[2]t_fifo):    #fifo_full,empty#
BEGIN
    FN FIFO_RAM = (STRING[16]bit:in,t_inp:wr_addr rd_addr,t_load:rw_fifo)   ->STRING[16]bit:
    RAM(b"0000000000000000").

    FN FULL = (t_inp:in)    ->t_fifo:ARITH IF in>1023 THEN 2   #fifo full#
                    ELSE1
                    FI.
    FN INCR = (t_inp:in)    ->t_inp:ARITH in+1.

      FN EMPTY = (t_inp:in)    ->t_fifo:ARITH IF in<0 THEN 2  #fifo empty#
                         ELSE  1
                         FI.
      FN DECR = (t_inp:in)   ->t_inp:ARITH in-1.

MAKE DFF{t_inp}:address,
```

```
    FIFO_RAM:ram.

LET   next =    CASE direction
      OFforward: CASE fifo_write
            OFwrite:INCR address
            ELSE address
            ESAC,
         inverse:CASE fifo_read
            OFread:INCR address
            ELSE   address
                  ESAC
      ESAC.


JOIN  (ck,reset,next,inp/0)        ->address,
   (buffer_in ,address,address,CASE direction
            OF  inverse:read,
              forward:fifo_write
            ESAC) ->ram.

OUTPUT     (ram,(FULL address,EMPTY address))
END.

FN TEST_PALMAS = (bool:ck,t_reset:reset,t_direction:direction,t_intra:intra_inter,t_channel_factor:channel_factor,
       t_input:q_int,t_quant:quant_norm,t_result:threshold comparison)

      ->(STRING[16]bit,#buffer_out#[2]t_load#fifo_read fifo_write#,bool,bool,t_int32):

BEGIN
```

```
MAKE SPARC_MEM:new old_inv old_forw,
    FIFO:fifo,
    PALMAS:palmas_inv palmas_forw.

LET     col_length = (IN_TO_S{9} input/31)[2],

    row_length= (IN_TO_S{9} input/31)[2],

    ximage_string = (IN_TO_S{9} input/32)[2],

    yimage_string = (IN_TO_S{9} input/32)[2],

    yimage_string_3 = (IN_TO_S{9} input/80)[2],

    pro_forw = palmas_forw[1],

    pro_inv = palmas_inv[1],

    forw_frame_done = palmas_forw[7],

    inv_frame_done = palmas_inv[7],

    cycle = palmas_inv[8],

    old_equal = CASE cycle
            OF  data_cycle:old_forw EQ palmas_inv[1]
            ELSE t
            ESAC.

JOIN
```

```
#fix fifo full/empty logic later#
(ck,reset,forward,intra_inter,channel_factor,q_int,quant_norm,b"000000000000000",new,old_forw, threshold,comparison,
#fifo[2][1],fifo[2][2]#ok_fifo,ok_fifo, col_length,row_length,ximage_string,yimage_string, yimage_string_3)
                              ->palmas_forw,

(ck,reset,inverse,intra_inter,channel_factor,q_int,quant_norm,fifo[1],new,old_inv, threshold,comparison,
#fifo[2][1],fifo[2][2]#ok_fifo,ok_fifo, col_length,row_length,ximage_string,yimage_string, yimage_string_3)
                              ->palmas_inv,

#old forward mem, on forward use as normal, on inverse read values to compare with inverse#
(pro_forw,CASE direction
   OF   forward:palmas_forw[2],
     inverse:palmas_inv[2]
   ESAC, CASE direction
     OF   forward:palmas_forw[2],
        inverse:palmas_inv[2]
     ESAC,CASE direction
        OF   forward:palmas_forw[4][1],
           inverse:read
        ESAC)         ->old_forw,

(palmas_inv[1],palmas_inv[2],palmas_inv[2],CASE direction
               OF  forward:read,
                inverse:palmas_inv[4][1]
               ESAC)     ->old_inv,

#(input/0,palmas_forw[2],palmas_forw[2],palmas_forw[3][1])        ->new,#
(input/0,CASE direction
     OF   forward:palmas_forw[2],
        inverse:palmas_inv[2]
     ESAC, CASE direction
```

```
    OF   forward:palmas_forw[2],
         inverse:palmas_inv[2]
    ESAC,CASE direction
         OF   forward:palmas_forw[3][1],
              inverse:read
         ESAC)                    ->new,


(ck,reset,CASE direction
     OF   inverse:b"0000000000000000",
          forward:palmas_forw[5]
     ESAC          ,direction,palmas_inv[6][1],palmas_forw[6][2])     ->fifo.

OUTPUT (palmas_forw[5],palmas_forw[6],palmas_forw[7],old_equal,RMS(ck,reset,cycle,old_inv,new) )
END.

#test for palmas chip#
TYPE t_int32 = NEW int32/(-2147483000..2147483000).

FN RMS = (bool:ck,t_reset:reset,t_cycle:cycle, t_input:old new) ->t_int32:
BEGIN

FN I_32 = (t_input:in)  ->t_int32:ARITH in.
FN DV = (t_int32:a b)   ->t_int32:ARITH a%b.
FN PL = (t_int32:a b)   ->t_int32:ARITH a+b.
FN MI = (t_int32:a b)   ->t_int32:ARITH a-b.
FN TI = (t_int32:a b)   ->t_int32:ARITH a*b.

MAKE   DFF_INIT(t_int32):old_error.
LET err = I_32old MI I_32new,
    err2 = (errTIerr) PL old_error.
```

```
JOIN    (ck,reset,CASE cycle
            OF     data_cycle:write
            ELSE   read
            ESAC,err2,int32/0)          ->old_error.

OUTPUT old_error
END.
FN EQ = (t_input:a b)   ->bool:ARITH IF a=b THEN 2
                          ELSE 1
                          FI.

FN SPARC_MEM = (t_input:in,t_sparc_addr:wr_addr,t_sparc_addr:rd_addr,t_load:rw_sparc#,t_cs:cs#)->t_input:
            RAM(input/0).

FN FIFO_BIG        =(bool:ck,t_reset:reset,STRING[16]bit:buffer_in,t_direction:direction,t_load:fifo_read fifo_write)
                          ->(STRING[16]bit,[2]t_fifo):   #fifo_full,empty#
BEGIN
      FN  FIFO_RAM = (STRING[16]bit:in,t_sparc_addr:wr_addr rd_addr,t_load:rw_fifo)   ->STRING[16]bit:
      RAM(b"0000000000000000").

      FN FULL = (t_sparc_addr:in)    ->t_fifo:ARITH IF in>1023 THEN 2       #fifo full#
                          ELSE  1
                          FI.
      FN INCR = (t_sparc_addr:in)    ->t_sparc_addr:ARITH in+1.

      FN EMPTY = (t_sparc_addr:in)    ->t_fifo:ARITH IF in<0 THEN 2  #fifo empty#
                          ELSE  1
                          FI.
      FN DECR = (t_sparc_addr:in)    ->t_sparc_addr:ARITH in-1.
```

```
MAKE   DFF{t_sparc_addr}:address,
     FIFO_RAM:ram.
LET     next = CASE direction
          OF     forward: CASE fifo_write
                    OF     write:INCR address
                    ELSE   address
                    ESAC,
               inverse:CASE fifo_read
                    OF     read:INCR address
                    ELSE   address
                    ESAC
          ESAC.


JOIN   (ck,reset,next,addr/0)         ->address,
     (buffer_in ,address,address,CASE direction
                      OF  inverse:read,
                          forward:fifo_write
                      ESAC)     ->ram.

OUTPUT        (ram,(FULL address,EMPTY address))
END.

FN TEST_PALMAS = (bool:ck,t_reset:reset, bool:load_memory,t_direction:direction,t_intra:intra_inter,
       t_channel_factor:channel_factor,[4]t_quant:quant_norm,[4]t_result:threshold,
       t_input: col_length_in row_length_in ximage_string_in yimage_string_in,
       t_result:yimage_string_3_in)

       ->(bool#,t_int32#):
```

```
BEGIN

  FN NEW_ADDRESS = (t_sparc_addr:in)        ->t_sparc_addr: ARITH ((in +1) MOD 120000).

  MAKE SPARC_MEM:new old_inv old_forw,
     FIFO_BIG:fifo,
     PRBS11:prbs,
        DFF{t_sparc_addr}:address,
     PALMAS:palmas.

LET       col_length = (IN_TO_S{10} col_length_in)[2].

        row_length= (IN_TO_S{9} row_length_in)[2].

        ximage_string = (IN_TO_S{10} ximage_string_in)[2].

        yimage_string = (IN_TO_S{9} yimage_string_in)[2].

        yimage_string_3 = (I_TO_SC{11} yimage_string_3_in)[2].

     pro= palmas[1].

     random_data = BOOL_INT10 prbs.

     frame_done = palmas[7].

     cycle = palmas[8].

     old_equal = CASE cycle
```

```
                OF  data_cycle:old_forw EQ  palmas[1]
                ELSE t
                ESAC.


JOIN
#fix fifo full/empty logic later#
(ck,reset,direction,intra_inter,channel_factor,quant_norm,CASE direction
                        OFforward:b"0000000000000000"
                        ELSE fifo[1]
                        ESAC,new,CASE direction
                              OF forward:old_forw
                              ELSE old_inv
                              ESAC, threshold,
#fifo[2][1],fifo[2][2]#ok_fifo,ok_fifo, col_length,row_length,ximage_string,yimage_string, yimage_string_3)
                    ->palmas,
(ck,reset,(NEW_ADDRESS address) ,addr/0)                -> address,

(ck,reset)                                  ->prbs,

#old forward mem, on forward use as normal, on inverse read values to compare with inverse#
(CASE load_memory
OF     t:DFF{t_input}(ck,reset,random_data,input/0)
ELSE   palmas[1]
ESAC   , CASE load_memory
       OF     t:address
       ELSE       palmas[2]
       ESAC,           palmas[2], CASE load_memory
                           OF  t:write
                           ELSE      CASE direction
                                  OF    forward:palmas[4][1],
```

```
                                inverse:read
                        ESAC
                ESAC)                   ->old_forw,

(CASE load_memory
OF      t:DFF{t_input}(ck,reset,random_data,input/0)
ELSE   palmas[1]
ESAC    , CASE load_memory
        OF      t:address
        ELSE        palmas[2]
        ESAC;           palmas[2], CASE load_memory
                                OF  t:write
                                ELSE     CASE direction
                                        OF  forward:read,
                                            inverse:palmas[4][1]
                                        ESAC
                ESAC)                   ->old_inv,

(CASE load_memory
OF      t:random_data
ELSE   input/0
ESAC    , CASE load_memory
        OF      t:address
        ELSE        palmas[2]
        ESAC,           palmas[2], CASE load_memory
                                OF  t:write
                                ELSE     CASE direction
                                        OF  forward:palmas[3][1],
                                            inverse:read
                                        ESAC
                ESAC)                   ->new,
```

```
(ck,reset,CASE direction
      OF    inverse:b"0000000000000000",
            forward:palmas[5]
      ESAC          ,direction,palmas[6][1],palmas[6][2])      ->fifo.

OUTPUT (old_equal#,RMS(ck,reset,cycle,old_inv,new)# )
END.


#test for palmas chip#
TYPE t_int32 = NEW int32/(-2147483000..2147483000).

FN RMS = (bool:ck,t_reset:reset,t_cycle:cycle, t_input:old new)    ->t_int32:
BEGIN

FN I_32 = (t_input:in)  ->t_int32:ARITH in.
FN DV = (t_int32:a b)   ->t_int32:ARITH a%b.
FN PL = (t_int32:a b)   ->t_int32:ARITH a+b.
FN MI = (t_int32:a b)   ->t_int32:ARITH a-b.
FN TI = (t_int32:a b)   ->t_int32:ARITH a*b.

MAKEDFF_INIT{t_int32}:old_error.

LET  err = I_32old MI I_32new,
   err2 = (errTIerr) PL  old_error.

JOIN  (ck,reset,CASE cycle
       OFdata_cycle:write
       ELSE read
       ESAC,err2,int32/0)      ->old_error.
```

```
OUTPUT old_error
END.

FN EQ = (t_input:a b)   ->bool:ARITH IF a=b THEN 2
                    ELSE 1
                    FI.

FN SPARC_MEM = (t_input:in,t_sparc_addr:wr_addr,t_sparc_addr:rd_addr,t_load:rw_sparc#,t_cs:cs#)->t_input:
      RAM(input/0).

FN FIFO     =(bool:ck,t_reset:reset,STRING[16]bit:buffer_in,t_direction:direction,t_load:fifo_read fifo_write)
                  ->(STRING[16]bit,[2]t_fifo):    #fifo_full,empty#
BEGIN
    FN  FIFO_RAM = (STRING[16]bit:in,t_inp:wr_addr rd_addr,t_load:rw_fifo)    ->STRING[16]bit:
    RAM(b"0000000000000000").

    FN FULL = (t_inp:in)    ->t_fifo:ARITH IF in>1023 THEN 2   #fifo full#
                    ELSE 1
                    FI.
    FN INCR = (t_inp:in)    ->t_inp:ARITH in+1.

      FN EMPTY = (t_inp:in)    ->t_fifo:ARITH IF in<0 THEN 2  #fifo empty#
                          ELSE  1
                          FI.
      FN DECR = (t_inp:in)    ->t_inp:ARITH in-1.


MAKE DFF{t_inp}:address,
  FIFO_RAM:ram.
```

```
LET    next =    CASE direction
       OFforward: CASE fifo_write
              OFwrite:INCR address
              ELSE address
              ESAC,
          inverse:CASE fifo_read
              OFread:INCR address
              ELSE   address
                     ESAC
       ESAC.


JOIN  (ck,reset,next,inp/0)        ->address,
    (buffer_in ,address,address,CASE direction
                  OF  inverse:read,
                   forward:fifo_write
                  ESAC) ->ram.

OUTPUT     (ram,(FULL address,EMPTY address))
END.

FN TEST_PALMAS = (bool:ck,t_reset:reset, bool:load_memory,t_direction:direction,t_intra:intra_inter,t_channel_factor:channel_factor,
        t_input:q_int,t_quant:quant_norm,t_result:threshold comparison)

       ->(bool,t_int32):


BEGIN

   FN NEW_ADDRESS = (t_sparc_addr:in)       ->t_sparc_addr: ARITH ((in +1) MOD 120000).
```

```
MAKE SPARC_MEM:new old_inv old_forw,
    FIFO:fifo,
    PRBS11:prbs,
    DFF{t_sparc_addr}:address,
    PALMAS:palmas.

LET    col_length = (IN_TO_S{10} input/31)[2],

    row_length= (IN_TO_S{9} input/31)[2],

    ximage_string = (IN_TO_S{10} input/32)[2],

    yimage_string = (IN_TO_S{9} input/32)[2],

    yimage_string_3 = (I_TO_SC{11} result/80)[2],

    pro= palmas[1],

    random_data = BOOL_INT10 prbs,

    frame_done = palmas[7],

    cycle = palmas[8],

    old_equal = CASE cycle
            OF   data_cycle:old_forw EQ  palmas[1]
            ELSE t
            ESAC.
```

```
JOIN
#fix fifo full/empty logic later#
(ck,reset,direction,intra_inter,channel_factor,q_int,quant_norm,fifo[1],new,CASE direction
                                        OF forward:old_forw
                                        ELSE old_inv
                                        ESAC, threshold,comparison,
#fifo[2][1],fifo[2][2]#ok_fifo,ok_fifo, col_length,row_length,ximage_string,yimage_string, yimage_string_3)
                               ->palmas,

(ck,reset,(NEW_ADDRESS address) ,addr/0)            -> address,

(ck,reset)                      ->prbs,

#old forward mem, on forward use as normal, on inverse read values to compare with inverse#
(CASE load_memory
OFt:DFF{t_input}(ck,reset,random_data,input/0)
ELSE palmas[1]
ESAC , CASE load_memory
     OF     t:address
     ELSE   palmas[2]
     ESAC,       palmas[2], CASE load_memory
                   OF  t:write
                   ELSE  CASE direction
                      OF   forward:palmas[4][1],
                           inverse:read
                      ESAC
                 ESAC)          ->old_forw,

(CASE load_memory
OF     t:DFF{t_input}(ck,reset,random_data,input/0)
ELSE   palmas[1]
```

```
ESAC   , CASE load_memory
        OF     t:address
        ELSE       palmas[2]
        ESAC,              palmas[2], CASE load_memory
                                OF  t:write
                                ELSE      CASE direction
                                          OF   forward:palmas[4][1],
                                               inverse:read
                                          ESAC
                           ESAC)               ->old_inv,

(CASE load_memory
OF     t:random_data
ELSE   input/0
ESAC   , CASE load_memory
        OF     t:address
        ELSE       palmas[2]
        ESAC,              palmas[2], CASE load_memory
                                OF  t:write
                                ELSE      CASE direction
                                          OF   forward:palmas[3][1],
                                               inverse:read
                                          ESAC
                           ESAC)               ->new,


(ck,reset,CASE direction
     OF    inverse:b"0000000000000000",
           forward:palmas[5]
```

```
    ESAC          ,direction,palmas[6][1],palmas[6][2])     ->fifo.
OUTPUT (old_equal,RMS(ck,reset,cycle,old_inv,new) )
END.
```

APPENDIX C

7/22/93 3:39 PM Engineering:KlicsCode:CompPict:Top.a

```
*--------------------------------
*
*   © Copyright 1993 KLICS Limited
*   All rights reserved.
*
*   Written by: Adrian Lewis
*
*--------------------------------
*
*   680X0 Fast Top Octave
*
*--------------------------------
        seg         'klics'

        macro
        TOPX        &DG, &HG, &old, &XX
*
        swap        &HG                     ; HG=G1H0
        move.w      &DG,&XX                 ; XX=G0
        neg.w       &DG                     ; DG=D(-G0)
        add.w       &HG,&DG                 ; DG=DD
        add.w       &XX,&HG                 ; HG=G1D
        swap        &HG                     ; HG=DG1
        move.l      &DG,&old                ; save DD
*
        endm
*--------------------------------
        macro
        TOPY        &HG0, &new0, &HG1, &new1, &XX
*
        move.l      &new0,&XX               ; read HG
        move.l      &new1,&HG1              ; read HG
        move.l      &HG1,&HG0               ; copy HG
        add.l       &XX,&HG1                ; new1=H1G1
        sub.l       &XX,&HG0                ; new0=H0G0
*
        endm
*--------------------------------
        macro
        TOPBLOCK    &DG0, &HG0, &new0, &old0, &DG1, &HG1, &new1, &old1, &XX
*
        TOPY        &HG0,&new0,&HG1,&new1,&XX
        TOPX        &DG0,&HG0,&old0,&XX
        TOPX        &DG1,&HG1,&old1,&XX
*
        endm
*--------------------------------
        macro
        TOPH        &DG, &HG, &new, &old, &XX
*
        move.l      &new,&HG
        TOPX        &DG,&HG,&old,&XX
*
        endm
*--------------------------------
        macro
        TOPE        &DG, &old, &XX
*
        move.l      &DG,&XX                 ; XX=DG
        swap        &XX                     ; XX=GD
        move.w      &XX,&DG                 ; DG=DD
        move.l      &DG,&old                ; save DD
*
```

```
        endm
*-------------------------------
TopBwd  FUNC    EXPORT
*
PS      RECORD      8
src     DS.L        1
dst     DS.L        1
width   DS.L        1
height  DS.L        1
        ENDR
*
        link        a6,#0                       ; no local variables
        movem.l     d4-d7/a3-a5,-(a7)           ; store registers
*
        movea.l     PS.src(a6),a0               ; read src
        move.l      PS.height(a6),d7            ; read height
        move.l      PS.width(a6),d6             ; read width
*       move.l      a0,a1
        move.l      PS.dst(a6),a1               ; read dst

        move.l      d6,d5                       ; inc = width
        add.l       d5,d5                       ; inc*=2
        move.l      d5,a4                       ; save inc

        lsr.l       #1,d7                       ; height/=2
        subq.l      #2,d7                       ; height-=2

        lsr.l       #2,d6                       ; width/=4
        subq.l      #2,d6                       ; width-=2

        move.l      d6,d5                       ; ccount=width
        move.l      (a0)+,d0                    ; d0=*new0++

@do1    TOPH        d0,d1,(a0)+,(a1)+,d4
        TOPH        d1,d0,(a0)+,(a1)+,d4
        dbf         d5,@do1                     ; while -1!=--ccount
        TOPH        d0,d1,(a0)+,(a1)+,d4
        TOPE        d1,(a1)+,d4

@do2    move.l      a0,a2                       ; new0=new1
        move.l      a1,a3                       ; old0=old1
        adda.l      a4,a0                       ; new1+=inc
        adda.l      a4,a1                       ; old1+=inc
        move.l      d6,d5                       ; ccount=width
        TOPY        d2,(a2)+,d0,(a0)+,d4

@do3    TOPBLOCK    d2,d3,(a2)+,(a3)+,d0,d1,(a0)+,(a1)+,d4
        TOPBLOCK    d3,d2,(a2)+,(a3)+,d1,d0,(a0)+,(a1)+,d4
        dbf         d5,@do3                     ; while -1!=--ccount

        TOPBLOCK    d2,d3,(a2)+,(a3)+,d0,d1,(a0)+,(a1)+,d4
        TOPE        d1,(a1)+,d4
        TOPE        d3,(a3)+,d4
        dbf         d7,@do2                     ; while -1!=--height

        move.l      d6,d5                       ; ccount=width
        add.l       #1,d5                       ; d0=*new0++

@do4    move.l      (a3)+,(a1)+                 ; copy prev line
        move.l      (a3)+,(a1)+
        dbf         d5,@do4                     ; while -1!=--ccount

        movem.l     (a7)+,d4-d7/a3-a5           ; restore registers
```

```
        unlk        a6                      ; remove locals
        rts                                 ; return
*
        ENDFUNC
*-----------------------------------
        END
```

Engineering:KlicsCode:CompPict:Table.a

```
*--------------------------------
*
*   © Copyright 1993 KLICS Ltd.
*   All rights reserved.
*
*--------------------------------
*
*   680X0 Table Lookup RGB/YUV code
*
*--------------------------------

        machine     MC68030
        seg         'klics'

        if  &TYPE('seg')≠'UNDEFINED' then
        seg         &seg
        endif

MKTABLE FUNC    EXPORT
*
PS      RECORD      8
Table   DS.L        1
        ENDR
*
        link        a6,#0                   ;
        movem.l     d4-d7/a3-a5,-(a7)       ; store registers
*
        move.l      PS.Table(a6),a0         ;Table is (long)(2U+512) (long)(512-(6
        clr.l       d0                      ;U value
@MakeLoop
        move.w      #512,d1                 ;512

        move.l      d0,d2                   ;U
        move.w      d2,d3                   ;U

        add.w       d2,d2                   ;2U
        add.w       d1,d2                   ;2U + 512

        lsr.w       #2,d2
        move.w      d2,(a0)+                ;Place 1st word
        move.w      d2,(a0)+                ;Place 2nd word

        add.w       d3,d3                   ;2U
        move.w      d3,d2                   ;2U
        add.w       d3,d3                   ;4U
        add.w       d2,d3                   ;6U
        asr.w       #4,d3                   ;6U/16
        sub.w       d3,d1                   ;512 - (6U/16)

        lsr.w       #2,d1
        move.w      d1,(a0)+                ;Place 1st word
        move.w      d1,(a0)+                ;Place 2nd word

        add.w       #1,d0
        cmp.w       #$0200,d0
        bne         @MakeLoop

        move.l      #$00000200,d0           ;U value
        clr.l       d4
@MakeNegLoop
        move.w      #512,d1                 ;512

        move.w      d0,d2                   ;U
```

Engineering:KlicsCode:CompPict:Table.a

```
        or.w        #$FC00,d2
        move.w      d2,d3                   ;U

        add.w       d2,d2                   ;2U
        add.w       d1,d2                   ;2U + 512

        asr.w       #2,d2
        move.w      d2,(a0)+                ;Place 1st word
        move.w      d2,(a0)+                ;Place 2nd word
        add.w       d3,d3                   ;2U
        move.w      d3,d2                   ;2U
        add.w       d3,d3                   ;4U
        add.w       d2,d3                   ;6U
        asr.w       #4,d3                   ;6U/16
        sub.w       d3,d1                   ;512 - (6U/16)

        asr.w       #2,d1
        move.w      d1,(a0)+                ;Place 1st word
        move.w      d1,(a0)+                ;Place 2nd word

        add.l       #1,d0
        add.l       #1,d4
        cmp.w       #$0200,d4
        bne         @MakeNegLoop
*
        movem.l     (a7)+,d4-d7/a3-a5       ; restore registers
        unlk        a6                      ; remove locals
        rts                                 ; return
*
        ENDFUNC
*---------------------------------
        macro
        FIXOV       &V, &SP1, &SP2
*
        move.w      &V,&SP1
        clr.b       &SP1
        andi.w      #$3FFF,&SP1
        sne         &SP1
        btst        #13,&SP1
        seq         &SP2
        or.b        &SP1,&V
        and.w       &SP2,&V
        swap        &V
        move.w      &V,&SP1
        clr.b       &SP1
        andi.w      #$3FFF,&SP1
        sne         &SP1
        btst        #13,&SP1
        seq         &SP2
        or.b        &SP1,&V
        and.w       &SP2,&V
        swap        &V
*
        endm
*---------------------------------
        if  &TYPE('seg')≠'UNDEFINED' then
        seg         &seg
        endif

YUV2RGB4    FUNC        EXPORT
*
PS      RECORD      8
Table   DS.L        1
```

Engineering:KlicsCode:CompPict:Table.a

```
pixmap  DS.L     1
Y       DS.L     1
U       DS.L     1
V       DS.L     1
area    DS.L     1
width   DS.L     1
cols    DS.L     1
        ENDR
*
LS      RECORD   0,DECR
inc     DS.L     1
width   DS.L     1
fend    DS.L     1
count   DS.L     1
LSize   EQU      *
        ENDR
*
*void YUVtoRGB(Ptr TablePtr,long *pixmap,short *Yc,short *Uc,short *Vc,long area,l
*{
*long       inc,lwidth,fend,count;
;       a0 - Y0, a1 - Y1, a2 - U, a3 - V, a4 - pm0, a5 - pm1
;       d0..6 - used, d7 - count

        link        a6,#LS.LSize             ; save locals
        movem.l     d0-d7/a0-a5,-(a7)        ; store registers

        move.l      PS.pixmap(a6),a4         ; pm0=pixmap
        move.l      a4,a5                    ; pm1=pm0
        move.l      PS.Y(a6),a0              ; Y0=Yc
        move.l      a0,a1                    ; Y1=Y0
        move.l      PS.U(a6),a2              ; U=Uc
        move.l      PS.V(a6),a3              ; V=Vc
        move.l      PS.area(a6),d7           ; fend=area
        lsl.l       #2,d7                    ; fend<<=2
        add.l       a4,d7                    ; fend+=pm0
        move.l      d7,LS.fend(a6)           ; save fend
        move.l      PS.width(a6),d5          ; width=width
        move.l      d5,d7                    ; count=width
        asr.l       #1,d7                    ; count>>=1
        subq.l      #1,d7                    ; count-=1
        move.l      d7,PS.width(a6)          ; save width

        add.l       d5,d5                    ; width*=2
        add.l       d5,a1                    ; Y1+=width
        add.l       d5,d5                    ; width*=2
        move.l      d5,LS.width(a6)          ; save width

        move.l      PS.cols(a6),d4           ; inc=cols
        lsl.l       #2,d4                    ; inc<<=2
        add.l       d4,a5                    ; pm1+=inc
        add.l       d4,d4                    ; cols*=2
        sub.l       d5,d4                    ; inc now 2*cols-width bytes
        move.l      d4,LS.inc(a6)            ; save inc

        move.l      a6,-(sp)
        move.l      PS.Table(a6),a6

; Colors wanted are:
;   RED     = (Y + 2V + 512) / 4                    UTable part is for  (2V + 512)
;   GREEN   = (Y - V + 512 - (6U/16)) / 4           UTable part is for  (512 - (6U
;   BLUE    = (Y + 2U + 512) / 4                    UTable part is for  (2U + 512)
;
@do      ; uv2rgb(*U++,*V++)
```

Engineering:KlicsCode:CompPict:Table.a

```
        d1 - ra, d2 - ga, d3 - ba,    d4 - rb, d5 - gb/512, d6 - bb

        move.w      (a2)+,d2                ; U
        beq         @DoQuickU
        and.w       #$03FF,d2
        move.l      (a6,d2.w*8),d3          ;BLUE,Get (2U + 512)/4 for Blue = (Y +
        move.l      d3,d6                   ;Dup for second pair
        move.l      4(a6,d2.w*8),d5         ;GREEN, Get (512 - (6U/16))/4 for Gree
@DidQuickU
        move.w      (a3)+,d1                ; V
        beq         @DoQuickV               ;if zero then handle using the quick m
        move.w      d1,d4
        asr.w       #2,d1
        sub.w       d1,d5                   ;GREEN, Get (512 - (6U/16) - V)/4 for
        move.w      d5,d2
        swap        d5
        move.w      d2,d5
        move.l      d5,d2                   ;Dup for second pair

        and.w       #$03FF,d4
        move.l      (a6,d4.w*8),d4          ;RED, Get (2V + 512)/4 for Red = (Y +
        move.l      d4,d1
        bra         @TestEnd

@DoQuickU
        move.l      #$00800080,d3           ;BLUE,Get (2U + 512)/4 for Blue = (Y +
        move.l      d3,d6                   ;Dup for second pair
        move.l      d3,d5                   ;GREEN, Get (512 - (6U/16))/4 for Gree
        bra         @DidQuickU

@DoQuickV
        move.l      d5,d2                   ;GREEN, Get (512 - (6U/16) - V)/4 for
        move.l      #$00800080,d4           ;RED, Get (2V + 512)/4 for Red = (Y +
        move.l      d4,d1                   ;Dup for second pair

@TestEnd

        ; add Ya to RGB values - FETCHY (a0)+,d0,d1,d2,d3
        move.l      (a0)+,d0                ;Y
        asr.w       #2,d0
        swap        d0
        asr.w       #2,d0
        swap        d0                      ;Y is       -128 to +127
        add.l       d0,d1                   ;RED, Get (Y+ 2V + 512) for Red = (Y +
        add.l       d0,d2                   ;GREEN, Get (Y + (512 - (6U/16)) - V)
        add.l       d0,d3                   ;BLUE,Get (Y + (2U + 512) for Blue = (

        ; add Yb to RGB values - FETCHY2 (a1)+,d0,d4,d5,d6
        move.l      (a1)+,d0                ;Y
        asr.w       #2,d0
        swap        d0
        asr.w       #2,d0
        swap        d0                      ;Y is       -128 to +127
        add.l       d0,d4                   ;RED, Get (Y+ 2V + 512) for Red = (Y +
        add.l       d0,d5                   ;GREEN, Get (Y + (512 - (6U/16)) - V)
        add.l       d0,d6                   ;BLUE,Get (Y + (2U + 512) for Blue = (

        move.l      d1,d0
        or.l        d4,d0

        or.l        d2,d0
        or.l        d3,d0
        or.l        d5,d0
```

```
        or.l        d6,d0

        and.l       #$FF00FF00,d0
        bne         @over                   ; if overflow

@ok     ; save RGBa - MKRGB d1,d2,d3,(a4)+
        lsl.l       #8,d2                   ; G=G0G0 (12)
        or.l        d3,d2                   ; G=GBGB (12)
        move.l      d1,d3                   ; B=0R0R (12)
        swap        d3                      ; B=0R0R (21)
        move.w      d2,d3                   ; B=0RGB (2)
        swap        d2                      ; G=GBGB (21)
        move.w      d2,d1                   ; R=0RGB (1)
        move.l      d1,(a4)+                ; *RGB++=rgb (1)
        move.l      d3,(a4)+                ; *RGB++=rgb (2)

        ; save RGBb - MKRGB d4,d5,d6,(a5)+
        lsl.l       #8,d5                   ; G=G0G0 (12)
        or.l        d6,d5                   ; G=GBGB (12)
        move.l      d4,d6                   ; B=0R0R (12)
        swap        d6                      ; B=0R0R (21)
        move.w      d5,d6                   ; B=0RGB (2)
        swap        d5                      ; G=GBGB (21)
        move.w      d5,d4                   ; R=0RGB (1)
        move.l      d4,(a5)+                ; *RGB++=rgb (1)
        move.l      d6,(a5)+                ; *RGB++=rgb (2)

        dbf         d7,@do                  ; while

        move.l      (sp)+,a6

        adda.l      LS.inc(a6),a4           ; pm0+=inc
        adda.l      LS.inc(a6),a5           ; pm1+=inc
        adda.l      LS.width(a6),a0         ; Y0+=width
        exg.l       a0,a1                   ; Y1<->Y0
        move.l      PS.width(a6),d7         ; count=width
        cmpa.l      LS.fend(a6),a4          ; pm0<fend
        blt.w       @do2                    ; while

        movem.l     (a7)+,d0-d7/a0-a5       ; restore registers
        unlk        a6                      ; remove locals
        rts

@do2
        move.l      a6,-(sp)
        move.l      PS.Table(a6),a6
        bra         @do                     ; return


@FixIt
        btst        #31,d0                  ;See if upper word went negative
        beq         @D1TopNotNeg
        and.l       #$0000FFFF,d0           ;Pin at zero
@D1TopNotNeg
        btst        #24,d0                  ;See if upper word went too positive
        beq         @D1TopNotPos
        and.l       #$0000FFFF,d0           ;Mask old data out
        or.l        #$00FF0000,d0           ;New data is maxed
@D1TopNotPos

        btst        #15,d0                  ;See if lower word went negative
        beq         @D1BotNotNeg
```

Engineering:KlicsCode:CompPict:Table.a

```
        and.l       #$FFFF0000,d0          ;Pin at zero
@D1BotNotNeg
        btst        #8,d0                  ;See if lower word went too positive
        beq         @D1BotNotPos
        and.l       #$FFFF0000,d0          ;Mask old data out
        or.l        #$000000FF,d0          ;New data is maxed
@D1BotNotPos
        rts

@over
        move.l      d1,d0
        bsr         @FixIt
        move.l      d0,d1

        move.l      d2,d0
        bsr         @FixIt
        move.l      d0,d2

        move.l      d3,d0
        bsr         @FixIt
        move.l      d0,d3

        move.l      d4,d0
        bsr         @FixIt
        move.l      d0,d4

        move.l      d5,d0
        bsr         @FixIt
        move.l      d0,d5

        move.l      d6,d0
        bsr         @FixIt
        move.l      d0,d6

        bra         @ok

        ENDFUNC
*-----------------------------------
        END
```

Engineering:KlicsCode:CompPict:KlicsUtil.a

```
*--------------------------------
*
*   © Copyright 1993 KLICS Limited
*   All rights reserved.
*
*   Written by: Adrian Lewis
*
*--------------------------------
*
*   68000 Klics Utilities
*
*--------------------------------
        seg     'klics'

KLCopy  FUNC    EXPORT
*
*   KLCOPY(short *src, short *dst, int area);
*
PS      RECORD      8
src     DS.L        1
dst     DS.L        1
end     DS.L        1
        ENDR
*
        link        a6,#0               ; no local variables
*
        move.l      PS.src(a6),a0       ; short *src
        move.l      PS.dst(a6),a1       ; short *dst
        move.l      PS.end(a6),d3       ; long area
        lsr.l       #4,d3               ; in words(x8)
        subq.l      #1,d3               ; area-=1
@do     move.l      (a0)+,(a1)+         ; *dst++=*src++
        move.l      (a0)+,(a1)+         ; *dst++=*src++
        move.l      (a0)+,(a1)+         ; *dst++=*src++
        move.l      (a0)+,(a1)+         ; *dst++=*src++
        move.l      (a0)+,(a1)+         ; *dst++=*src++
        move.l      (a0)+,(a1)+         ; *dst++=*src++
        move.l      (a0)+,(a1)+         ; *dst++=*src++
        move.l      (a0)+,(a1)+         ; *dst++=*src++
        dbf         d3,@do              ; if -1!=--area goto do
*
        unlk        a6                  ; remove locals
        rts                             ; return
*
        ENDFUNC
*--------------------------------
KLHalf  FUNC    EXPORT
*
*   KLHALF(short *src, short *dst, long width, long height);
*   Dimensions of dst (width, height) are half that of src
*
PS      RECORD      8
src     DS.L        1
dst     DS.L        1
width   DS.L        1
height  DS.L        1
        ENDR
*
        link        a6,#0               ; no local variables
        movem.l     d4,-(a7)            ; store registers
*
        move.l      PS.src(a6),a0       ; short *src
        move.l      PS.dst(a6),a1       ; short *dst
```

Engineering:KlicsCode:CompPict:KlicsUtil.a

```
        move.l      PS.width(a6),d2         ; long width
        move.l      PS.height(a6),d3        ; long height
        subq.l      #1,d3                   ; height-=1
@do_y   move.l      d2,d4                   ; count=width
        lsr.l       #2,d4                   ; count /= 2
        subq.l      #1,d4                   ; count-=1
@do_x   move.l      (a0)+,d0                ; d0=*src++
        move.w      (a0)+,d0                ; d2=*src++
        addq.l      #2,a0                   ; src+=1 short
        move.l      d0,(a1)+                ; *dst++=d0
        move.l      (a0)+,d0                ; d0=*src++
        move.w      (a0)+,d0                ; d2=*src++
        addq.l      #2,a0                   ; src+=1 short
        move.l      d0,(a1)+                ; *dst++=d0
        dbf         d4,@do_x                ; if -1!=--width goto do_x
        adda.l      d2,a0                   ; skip a quarter row
        adda.l      d2,a0                   ; skip a quarter row
        adda.l      d2,a0                   ; skip a quarter row
        adda.l      d2,a0                   ; skip a quarter row
        dbf         d3,@do_y                ; if -1!=--height goto do_y
*
        movem.l     (a7)+,d4                ; restore registers
        unlk        a6                      ; remove locals
        rts                                 ; return
*
        ENDFUNC
*-----------------------------------
KLZero  FUNC        EXPORT
*
*       KLZERO(short *data, int area);
*
PS      RECORD      8
data    DS.L        1
end     DS.L        1
        ENDR
*
        link        a6,#0                   ; no local variables
*
        move.l      PS.data(a6),a0          ; short *data
        move.l      PS.end(a6),d3           ; long area
        lsr.l       #3,d3                   ; in words(x4)
        subq.l      #1,d3                   ; area-=1
@do     clr.l       (a0)+                   ; *dst++=*src++
        clr.l       (a0)+                   ; *dst++=*src++
        clr.l       (a0)+                   ; *dst++=*src++
        clr.l       (a0)+                   ; *dst++=*src++
        dbf         d3,@do                  ; if -1!=--area goto do
*
        unlk        a6                      ; remove locals
        rts                                 ; return
*
        ENDFUNC
*-----------------------------------
CLEARA2 FUNC        EXPORT
*
        move.l      #0,a2
        rts
*
        END
```

Engineering:KlicsCode:CompPict:KlicsEncode.h

```
/********************************
 *
 * © Copyright 1993 KLICS Limited
 * All rights reserved.
 *
 * Written by: Adrian Lewis
 *
 ********************************/
typedef struct {
    int     bpf_in,     /* User - Bytes per frame in input stream */
            bpf_out,    /* User - Bytes per frame in output stream */
            buf_size;   /* User - Buffer size (bytes) */

    Boolean intra,      /* Calc - Compression mode intra/inter */
            auto_q,     /* User - Automatic quantization for rate control */
            buf_sw;     /* User - Theoretical buffer on/off */

    float   quant,      /* User - Starting quantiser value */
            thresh,     /* User - Threshold factor */
            compare,    /* User - Comparison factor */
            base[5];    /* User - Octave weighting factors */

    int     buffer,     /* Calc - Current buffer fullness (bytes) */
            prevbytes,  /* Calc - Bytes sent last frame */
            prevquact;  /* Calc - Quantisation/activity for last frame */

    double  tmp_quant;  /* Calc - Current quantiser value quant */
} KlicsEDataRec;

typedef struct {
    KlicsSeqHeader      seqh;
    KlicsFrameHeader    frmh;
    KlicsEDataRec       encd;
    Buffer              buf;
} KlicsERec, *KlicsE;
```

Engineering:KlicsCode:CompPict:KlicsDec2.a

```
*------------------------------------
*
*   © Copyright 1993 KLICS Limited
*   All rights reserved.
*
*   Written by: Adrian Lewis
*
*------------------------------------
*
*   680X0 KlicsDecode code
*   Fast code for:
*       3/2 octave input stream
*       2/1 octave output image
*
*------------------------------------

        seg         'klics'
        include     'Bits3.a'
        include     'Traps.a'

*------------------------------------
        machine     MC68030

**********************************
*
*   Data stream readers:
*
*   XDELTA, XVALUE, SKIPHUFF, XINT
*
**********************************
        macro
        XDELTA      &addr,&step,&ptr,&data,&bno,&spare
*
        buf_rinc    &ptr,&data,&bno             ;
        buf_get     &data,&bno                  ;
        beq.s       @quit                       ; if zero write
        moveq       #6,&spare                   ; set up count
        buf_get     &data,&bno                  ; read sign
        bne.s       @doneg                      ; if negative -> doneg

@dopos  buf_get     &data,&bno
        dbne        &spare,@dopos               ; if --spare!=-1
        bne.s       @fndpos

        move.l      &data,&spare                ; spare=data
        subq.b      #7,&bno                     ; bno-=6
        lsr.l       &bno,&spare                 ; spare>>=bno
        andi.w      #$007F,&spare               ; spare AND= mask
        add.w       #8,&spare                   ; spare+=9
        bra.s       @write

@fndpos neg.w       &spare                      ; bits-=bits
        addq.l      #7,&spare                   ; bits+=8
        bra.s       @write

@doneg  buf_get     &data,&bno
        dbne        &spare,@doneg               ; if --spare!=-1
        bne.s       @fndneg

        move.l      &data,&spare                ; spare=data
        subq.b      #7,&bno                     ; bno-=6
        lsr.l       &bno,&spare                 ; spare>>=bno
        andi.w      #$007F,&spare               ; spare AND= mask
```

```
        add.w         #8,&spare               ; spare+=9
        neg.w         &spare
        bra.s         @write

@fndneg subq.l        #7,&spare               ; level-=8

@write  lsl.w         &step,&spare            ; level<<=step
        swap          &step
        add.w         &step,&spare
        swap          &step
        add.w         &spare,&addr            ; *addr=delta
@quit
;
        endm

        macro
        XVAL0         &addr,&step,&ptr,&data,&bno,&spare
;
        clr.w         &spare
        buf_rinc      &ptr,&data,&bno         ;
        buf_get       &data,&bno              ;
        beq.s         @quit                   ; if zero write
        moveq         #6,&spare               ; set up count
        buf_get       &data,&bno              ; read sign
        bne.s         @doneg                  ; if negative -> doneg

@dopos  buf_get       &data,&bno
        dbne          &spare,@dopos           ; if --spare!=-1
        bne.s         @fndpos

        move.l        &data,&spare            ; spare=data
        subq.b        #7,&bno                 ; bno-=6
        lsr.l         &bno,&spare             ; spare>>=bno
        andi.w        #$007F,&spare           ; spare AND= mask
        add.w         #8,&spare               ; spare+=9
        bra.s         @write

@fndpos neg.w         &spare                  ; bits-=bits
        addq.l        #7,&spare               ; bits+=8
        bra.s         @write

@doneg  buf_get       &data,&bno
        dbne          &spare,@doneg           ; if --spare!=-1
        bne.s         @fndneg

        move.l        &data,&spare            ; spare=data
        subq.b        #7,&bno                 ; bno-=6
        lsr.l         &bno,&spare             ; spare>>=bno
        andi.w        #$007F,&spare           ; spare AND= mask
        add.w         #8,&spare               ; spare+=9
        neg.w         &spare
        bra.s         @write

@fndneg subq.l        #7,&spare               ; level-=8

@write  lsl.w         &step,&spare            ; level<<=step
        swap          &step
        add.w         &step,&spare
        swap          &step
        move.w        &spare,&addr            ; *addr=level
@quit
;
        endm
```

Engineering:KlicsCode:CompPict:KlicsDec2.a

```
        macro
        XVAL1       &addr,&step,&ptr,&data,&bno,&spare

        clr.w       &spare
        buf_rinc    &ptr,&data,&bno             ;
        buf_get     &data,&bno                  ;
        beq.s       @quit                       ; if zero write
        moveq       #6,&spare                   ; set up count
        buf_get     &data,&bno                  ; read sign
        bne.s       @doneg                      ; if negative -> doneg

@dopos  buf_get     &data,&bno
        dbne        &spare,@dopos
        bne.s       @fndpos                     ; if --spare!=-1

        move.l      &data,&spare                ; spare=data
        subq.b      #7,&bno                     ; bno-=6
        lsr.l       &bno,&spare                 ; spare>>=bno
        andi.w      #$007F,&spare               ; spare AND= mask
        add.w       #8,&spare                   ; spare+=9
        bra.s       @write

@fndpos neg.w       &spare                      ; bits-=bits
        addq.l      #7,&spare                   ; bits+=8
        bra.s       @write

@doneg  buf_get     &data,&bno
        dbne        &spare,@doneg
        bne.s       @fndneg                     ; if --spare!=-1

        move.l      &data,&spare                ; spare=data
        subq.b      #7,&bno                     ; bno-=6
        lsr.l       &bno,&spare                 ; spare>>=bno
        andi.w      #$007F,&spare               ; spare AND= mask
        add.w       #8,&spare                   ; spare+=9
        neg.w       &spare
        bra.s       @write

@fndneg subq.l      #7,&spare                   ; level-=8

@write  lsl.w       &step,&spare                ; level<<=step
@quit   move.w      &spare,&addr                ; *addr=level

        endm

        macro
        SKIPHUFF    &ptr,&data,&bno,&spare

        buf_get     &data,&bno                  ;
        beq.s       @quit                       ; if zero quit
        buf_get     &data,&bno                  ; skip sign
        moveq       #6,&spare                   ; set up count

@do     buf_get     &data,&bno
        dbne        &spare,@do
        bne.s       @end                        ; if --spare!=-1

        subq.b      #7,&bno                     ; bno-=6
@end    buf_rinc    &ptr,&data,&bno             ; fill buffer
@quit

        endm
```

Engineering:KlicsCode:CompPict:KlicsDec2.a

```
        macro
        XINTX       &bits,&addr,&step,&ptr,&data,&bno
*
*   Note: half_q is missing
*
        buf_rinc    &ptr,&data,&bno         ;
        move.l      &data,d0                ; result=data
        sub.b       &bits,&bno              ; d1-=bits-1
        subq.b      #1,&bno                 ; d1-=1
        lsr.l       &bno,d0                 ; result>>=bno
        clr.l       d1                      ; d1=0
        bset        &bits,d1                ; d1[bits]=1
        subq.l      #1,d1                   ; d1=mask
        btst        &bits,d0                ; sign?
        beq.s       @pos                    ; if positive goto pos
        and.l       d1,d0                   ; apply mask leaving level
        neg.l       d0                      ; level-=level
        bra.s       @cont                   ; goto cont
@pos    and.l       d1,d0                   ; apply mask leaving level
@cont   lsl.l       &step,d0                ; level<<=step
        move.w      d0,&addr                ; *addr=result
*
        endm

        macro
        XINT        &bits,&addr,&step,&ptr,&data,&bno
*
*   Hardware compatable version: sign mag(lsb->msb)
*
        buf_rinc    &ptr,&data,&bno         ;
        move.l      &data,d0                ; result=data
        sub.b       &bits,&bno              ; d1-=bits-1
        subq.b      #1,&bno                 ; d1-=1
        lsr.l       &bno,d0                 ; temp>>=bno
        clr.l       d1                      ; result=0
        swap        &bno                    ; use free word
        move.w      &bits,&bno              ; bno=bno.bits
        subq.w      #1,&bno                 ; count=bits-2
@shft   lsr.l       #1,d0                   ; shift msb from temp
        roxl.l      #1,d1                   ; into lsb of result
        dbf         &bno,@shft              ; for entire magnitude
        swap        &bno                    ; restore bno
        btst        #0,d0                   ; sign test
        beq.s       @pos                    ; if positive -> pos
        neg.l       d1                      ; result= -result
@pos    lsl.l       &step,d1                ; result<<=step
        move.w      d1,&addr                ; *addr=result
*
        endm

*******************************
*
*   Block data read/write:
*
*   VOID, STILL, SEND, LPFSTILL
*
*******************************

        macro
        VOID        &x_blk, &y_blk
*
        clr.w       (a2)
```

```
        addq.l      &x_blk,a2               ; caddr+=x_blk
        clr.w       (a2)
        adda.w      &y_blk,a2               ; caddr+=y_blk
        clr.w       (a2)
        addq.l      &x_blk,a2               ; caddr+=x_blk
        clr.w       (a2)
*
        endm

        macro
        STILL       &x_blk, &y_blk, &step
*
        XVAL0       (a2),&step,a0,d6,d7,d0
        addq.l      &x_blk,a2               ; caddr+=x_blk
        XVAL0       (a2),&step,a0,d6,d7,d0
        adda.w      &y_blk,a2               ; caddr+=y_blk
        XVAL0       (a2),&step,a0,d6,d7,d0
        addq.l      &x_blk,a2               ; caddr+=x_blk
        XVAL0       (a2),&step,a0,d6,d7,d0
*
        endm

        macro
        STILLSEND   &x_blk, &y_blk, &step
*
        XVAL1       (a2),&step,a0,d6,d7,d0
        addq.l      &x_blk,a2               ; caddr+=x_blk
        XVAL1       (a2),&step,a0,d6,d7,d0
        adda.w      &y_blk,a2               ; caddr+=y_blk
        XVAL1       (a2),&step,a0,d6,d7,d0
        addq.l      &x_blk,a2               ; caddr+=x_blk
        XVAL1       (a2),&step,a0,d6,d7,d0
*
        endm

        macro
        SEND        &x_blk,&y_blk,&step
*
        XDELTA      (a2),&step,a0,d6,d7,d0
        addq.l      &x_blk,a2               ; caddr+=x_blk
        XDELTA      (a2),&step,a0,d6,d7,d0
        adda.w      &y_blk,a2               ; caddr+=y_blk
        XDELTA      (a2),&step,a0,d6,d7,d0
        addq.l      &x_blk,a2               ; caddr+=x_blk
        XDELTA      (a2),&step,a0,d6,d7,d0
*
        endm

        macro
        LPFSTILL    &x_blk, &y_blk, &step, &bits
*
        XINT        &bits,(a2),&step,a0,d6,d7   ; ReadInt (at baddr)
        addq.l      &x_blk,a2                   ; caddr+=x_blk
        XINT        &bits,(a2),&step,a0,d6,d7   ; ReadInt
        adda.w      &y_blk,a2                   ; caddr+=y_blk
        XINT        &bits,(a2),&step,a0,d6,d7   ; ReadInt
        addq.l      &x_blk,a2                   ; caddr+=x_blk
        XINT        &bits,(a2),&step,a0,d6,d7   ; ReadInt
*
        endm

*********************************
*
```

Engineering:KlicsCode:CompPict:KlicsDec2.a

```
*   Data skipping:
*
*   SKIP4, STILLSKIP, SS_SKIP, SENDSKIP
*
*******************************

SKIP4   FUNC    EXPORT
*
        buf_rinc    a0,d6,d7            ; fill buffer
        SKIPHUFF    a0,d6,d7,d0
        SKIPHUFF    a0,d6,d7,d0
        SKIPHUFF    a0,d6,d7,d0
        SKIPHUFF    a0,d6,d7,d0
        rts
*
        ENDFUNC

STILLSKIP   FUNC    EXPORT
*
        buf_rinc    a0,d6,d7            ; BUF_INC
        buf_get     d6,d7               ; BUF_GET
        beq.s       @sk1                ; if 0 the STOP
        bsr         SKIP4
        buf_rinc    a0,d6,d7            ; BUF_INC
@sk1    buf_get     d6,d7               ; BUF_GET
        beq.s       @sk2                ; if 0 the STOP
        bsr         SKIP4
        buf_rinc    a0,d6,d7            ; BUF_INC
@sk2    buf_get     d6,d7               ; BUF_GET
        beq.s       @sk3                ; if 0 the STOP
        bsr         SKIP4
        buf_rinc    a0,d6,d7            ; BUF_INC
@sk3    buf_get     d6,d7               ; BUF_GET
        beq.s       @nxt                ; if 0 the STOP
        bsr         SKIP4
@nxt    rts
*
        ENDFUNC

SS_SKIP FUNC    EXPORT
*
        buf_rinc    a0,d6,d7            ; BUF_INC
        buf_get     d6,d7               ; BUF_GET
        beq.s       @sk1                ; if 0 then STOP
        buf_get     d6,d7               ; BUF_GET
        bne.s       @sk1                ; if 1 then VOID
        bsr         SKIP4
        buf_rinc    a0,d6,d7            ; BUF_INC
@sk1    buf_get     d6,d7               ; BUF_GET
        beq.s       @sk2                ; if 0 then STOP
        buf_get     d6,d7               ; BUF_GET
        bne.s       @sk2                ; if 1 then VOID
        bsr         SKIP4
        buf_rinc    a0,d6,d7            ; BUF_INC
@sk2    buf_get     d6,d7               ; BUF_GET
        beq.s       @sk3                ; if 0 then STOP
        buf_get     d6,d7               ; BUF_GET
        bne.s       @sk3                ; if 1 then VOID
        bsr         SKIP4
        buf_rinc    a0,d6,d7            ; BUF_INC
@sk3    buf_get     d6,d7               ; BUF_GET
        beq.s       @nxt                ; if 0 then STOP
        buf_get     d6,d7               ; BUF_GET
```

```
        bne.s       @nxt                    ; if 1 then VOID
        bsr         SKIP4
@nxt    rts
*
        ENDFUNC

SENDSKIP    FUNC    EXPORT
*
        buf_rinc    a0,d6,d7                ; BUF_INC

        buf_get     d6,d7                   ; BUF_GET
        beq.s       @sk1                    ; if 0 the STOP
        buf_get     d6,d7                   ; BUF_GET
        beq.s       @sk0                    ; if 0 then STILLSEND
        buf_get     d6,d7                   ; BUF_GET
        beq.s       @sk1                    ; if 0 then VOID

@sk0    bsr         SKIP4
        buf_rinc    a0,d6,d7                ; BUF_INC

@sk1    buf_get     d6,d7                   ; BUF_GET
        beq.s       @sk3                    ; if 0 the STOP
        buf_get     d6,d7                   ; BUF_GET
        beq.s       @sk2                    ; if 0 then STILLSEND
        buf_get     d6,d7                   ; BUF_GET
        beq.s       @sk3                    ; if 0 then VOID

@sk2    bsr         SKIP4
        buf_rinc    a0,d6,d7                ; BUF_INC

@sk3    buf_get     d6,d7                   ; BUF_GET
        beq.s       @sk5                    ; if 0 the STOP
        buf_get     d6,d7                   ; BUF_GET
        beq.s       @sk4                    ; if 0 then STILLSEND
        buf_get     d6,d7                   ; BUF_GET
        beq.s       @sk5                    ; if 0 then VOID

@sk4    bsr         SKIP4
        buf_rinc    a0,d6,d7                ; BUF_INC

@sk5    buf_get     d6,d7                   ; BUF_GET
        beq.s       @nxt                    ; if 0 then STOP
        buf_get     d6,d7                   ; BUF_GET
        beq.s       @sk6                    ; if 0 then STILLSEND
        buf_get     d6,d7                   ; BUF_GET
        beq.s       @nxt                    ; if 0 then VOID

@sk6    bsr         SKIP4
@nxt    rts
*
        ENDFUNC

*********************************
*
*   Octave Processing:
*
*   DOSTILL0, DOSEND0, DOSTILL1,
*   DOVOID1, DOSTILLSEND1, DOSEND1
*
*********************************

DOSTILL0    FUNC    EXPORT
*
```

```
        buf_rinc    a0,d6,d7                ; BUF_INC
        buf_get     d6,d7                   ; BUF_GET
        bne.s       @still                  ; if 1 the STILL
        rts

@still  move.l      a1,a2                   ; caddr=baddr

        STILL       #4,d5,d3

        XVAL0       (a2),d3,a0,d6,d7,d0
        addq.l      #4,a2                   ; caddr+=x_blk
        XVAL0       (a2),d3,a0,d6,d7,d0
        adda.w      d5,a2                   ; caddr+=y_blk
        XVAL0       (a2),d3,a0,d6,d7,d0
        addq.l      #4,a2                   ; caddr+=x_blk
        XVAL0       (a2),d3,a0,d6,d7,d0

        bsr         STILLSKIP
        rts

        ENDFUNC

DOSEND0 FUNC    EXPORT

        buf_rinc    a0,d6,d7                ; BUF_INC
        buf_get     d6,d7                   ; BUF_GET
        bne.s       @cont                   ; if 1 then continue
        rts

@cont   move.l      a1,a2                   ; caddr=baddr
        buf_get     d6,d7                   ; BUF_GET
        beq.w       @ss                     ; if 0 then STILLSEND
        buf_get     d6,d7                   ; BUF_GET
        beq.w       @vd                     ; if 0 then VOID

        SEND        #4,d5,d3

        XDELTA      (a2),d3,a0,d6,d7,d0
        addq.l      #4,a2                   ; caddr+=x_blk
        XDELTA      (a2),d3,a0,d6,d7,d0
        adda.w      d5,a2                   ; caddr+=y_blk
        XDELTA      (a2),d3,a0,d6,d7,d0
        addq.l      #4,a2                   ; caddr+=x_blk
        XDELTA      (a2),d3,a0,d6,d7,d0

        bsr         SENDSKIP
        rts

@ss     ;STILLSEND  #4,d5,d3

        XVAL1       (a2),d3,a0,d6,d7,d0
        addq.l      #4,a2                   ; caddr+=x_blk
        XVAL1       (a2),d3,a0,d6,d7,d0
        adda.w      d5,a2                   ; caddr+=y_blk
        XVAL1       (a2),d3,a0,d6,d7,d0
        addq.l      #4,a2                   ; caddr+=x_blk
        XVAL1       (a2),d3,a0,d6,d7,d0

        bsr         SS_SKIP
        rts

@vd     ;VOID       #4,d5
```

Engineering:KlicsCode:CompPict:KlicsDec2.a

```
        clr.w       (a2)
        addq.l      #4,a2               ; caddr+=x_blk
        clr.w       (a2)
        adda.w      d5,a2               ; caddr+=y_blk
        clr.w       (a2)
        addq.l      #4,a2               ; caddr+=x_blk
        clr.w       (a2)
        rts
*
        ENDFUNC

        macro
        DOSTILL1    &addr
*
        buf_get     d6,d7               ; BUF_GET
        beq.w       @next               ; if 0 the STOP

        move.l      a1,a2               ; caddr=baddr
        add.l       &addr,a2            ; caddr+=addrs[1]
        STILL       #4,d5,d4
        bsr         STILLSKIP
        buf_rinc    a0,d6,d7            ; BUF_INC
@next
*
        endm

        macro
        DOVOID1     &addr
*
        move.l      a1,a2               ; caddr=baddr
        add.l       &addr,a2            ; caddr+=addrs[1]
        VOID        #4,d5
*
        endm

        macro
        DOSTILLSEND1    &addr
*
        buf_get     d6,d7               ; BUF_GET
        beq.w       @next               ; if 0 the STOP
        move.l      a1,a2               ; caddr=baddr
        add.l       &addr,a2            ; caddr+=addrs[1]
        buf_get     d6,d7               ; BUF_GET
        beq.s       @ss                 ; if 0 then STILLSEND

        VOID        #4,d5
        bra         @next

@ss     STILLSEND   #4,d5,d4
        bsr         SS_SKIP
        buf_rinc    a0,d6,d7            ; BUF_INC
@next
*
        endm

DOSTILL2    FUNC    EXPORT
*
        buf_rinc    a0,d6,d7            ; BUF_INC
        buf_get     d6,d7               ; BUF_GET
        bne.s       @cont               ; if 1 the CONT
        rts

@cont   move.l      a1,a2               ; caddr=baddr
```

Engineering:KlicsCode:CompPict:KlicsDec2.a

```
        add.l       (a3),a2            ; caddr+=addrs[0]
        STILL       #8,d5,d3

        swap        d5
        exg         d4,a5

        buf_rinc    a0,d6,d7           ; BUF_INC
        DOSTILL1    4(a3)
        DOSTILL1    8(a3)
        DOSTILL1    12(a3)
        DOSTILL1    16(a3)

        swap        d5
        exg         d4,a5
        rts

        macro
        DOSEND1     &addr

        buf_get     d6,d7              ; BUF_GET
        beq.w       @next              ; if 0 the STOP
        move.l      a1,a2              ; caddr=baddr
        add.l       &addr,a2           ; caddr+=addrs[1]
        buf_get     d6,d7              ; BUF_GET
        beq.w       @ss                ; if 0 then STILLSEND
        buf_get     d6,d7              ; BUF_GET
        beq.w       @vd                ; if 0 then VOID

        SEND        #4,d5,d4
        bsr         SENDSKIP
        bra         @rinc

@vd     VOID        #4,d5
        bra         @next

@ss     STILLSEND   #4,d5,d4
        bsr         SS_SKIP
@rinc   buf_rinc    a0,d6,d7           ; BUF_INC
@next
*
        endm

DOSEND2 FUNC    EXPORT
*
        buf_rinc    a0,d6,d7           ; BUF_INC
        buf_get     d6,d7              ; BUF_GET
        bne.s       @cont              ; if 1 the CONT
@nxt    rts

@cont   move.l      a1,a2              ; caddr=baddr
        add.l       (a3),a2            ; caddr+=addrs[0]
        buf_get     d6,d7              ; BUF_GET
        beq.w       @ss                ; if 0 then STILLSEND
        buf_get     d6,d7              ; BUF_GET
        beq.w       @vd                ; if 0 then VOID

*** SEND ***

        SEND        #8,d1,d3

        buf_rinc    a0,d6,d7           ; BUF_INC
        DOSEND1     4(a3)
        DOSEND1     8(a3)
```

```
        DOSEND1     12(a3)
        DOSEND1     16(a3)
        rts

*** STILLSEND ***

@ss     STILLSEND   #8,d1,d3

        buf_rinc    a0,d6,d7            ; BUF_INC
        DOSTILLSEND1    4(a3)
        DOSTILLSEND1    8(a3)
        DOSTILLSEND1    12(a3)
        DOSTILLSEND1    16(a3)
        rts

*** VOID ***

@vd     VOID        #8,d1

        DOVOID1     4(a3)
        DOVOID1     8(a3)
        DOVOID1     12(a3)
        DOVOID1     16(a3)
        rts
*
        ENDFUNC

        macro
        UVSTILL0
*
*   Low_Pass
*
        move.l      a1,a2               ; caddr=baddr
        LPFSTILL    #4,d5,d2,d4
*
*   Sub-band gh
*
        addq.l      #2,a1               ; baddr+=2 (gh band)
        bsr         DOSTILL0
*
*   Sub-band hg
*
        subq.l      #2,a1               ; baddr-=2 (hh band)
        add.l       a4,a1               ; caddr+=1 row (hg band)
        bsr         DOSTILL0
*
*   Sub-band gg
*
        addq.l      #2,a1               ; baddr+=2 (gg band)
        bsr         DOSTILL0
        sub.l       a4,a1               ; caddr-=1 row (gh band)
        addq.l      #6,a1               ; (2+) addr[0]+=x_inc
*
        endm

        macro
        UVSEND0
*
*   Low_Pass
*
        buf_rinc    a0,d6,d7            ; BUF_INC
        buf_get     d6,d7               ; BUF_GET
        beq.w       @subs               ; if 0 then process subbands
```

Engineering:KlicsCode:CompPict:KlicsDec2.a

```
        move.l      a1,a2                       ; caddr=baddr
        SEND        #4.d5,d2
*
*   Sub-band gh
*
@subs   addq.l      #2,a1                       ; baddr+=2 (gh band)
        bsr         DOSENDO
*
*   Sub-band hg
*
        subq.l      #2,a1                       ; baddr-=2 (hh band)
        add.l       a4,a1                       ; caddr+=1 row (hg band)
        bsr         DOSENDO
*
*   Sub-band gg
*
        addq.l      #2,a1                       ; baddr+=2 (gg band)
        bsr         DOSENDO
        sub.l       a4,a1                       ; caddr-=1 row (gh band)
        addq.l      #6,a1                       ; (2+) addr[0]+=x_inc
*
        endm

********************************
*
*   Decoder functions:
*
*   Klics2D1Still, Klics2D1Send
*
********************************

Klics2D1Still   FUNC    EXPORT
*
*   Klics2D1Still(short *dst, long size_x, long size_y, long lpfbits, short *norms
*
PS      RECORD      8
dst     DS.L        1
size_x  DS.L        1
size_y  DS.L        1
lpfbits DS.L        1
norms   DS.L        1
ptr     DS.L        1
data    DS.L        1
bno     DS.L        1
        ENDR
*
LS      RECORD      0,DECR
x_lim   DS.L        1                   ; x counter termination     row_start+
x_linc  DS.L        1                   ; x termination increment   1 row
y_inc0  DS.L        1                   ; y counter increment       4 rows
y_incl  DS.L        1                   ; y counter increment       7 rows
y_lim   DS.L        1                   ; y counter termination     area
LSize   EQU         *
        ENDR
*
*   d0/d1 - spare
*   d2 - step 0 (HH)
*   d3 - step 0
*   d4 - lpfbits
*   d5 - y_blk
*   d6 - data    (bit stream)
*   d7 - bno     (bit pointer)
*
```

Engineering:KlicsCode:CompPict:KlicsDec2.a

```
*       a0 - ptr      (bit buffer)
*       a1 - baddr    (block address)
*       a2 - caddr    (coeff address)
*       a3 - x_lim
*       a4 - x_linc
*       a5 - y_inc0
*
        link        a6,#LS.LSize              ; locals
        movem.l     d4-d7/a3-a5,-(a7)         ; store registers
*
*       Load Bit Buffer
*
        move.l      PS.data(a6),a0            ; a0=&data
        move.l      (a0),d6                   ; data=*a0
        move.l      PS.bno(a6),a0             ; a0=&mask
        move.l      (a0),d7                   ; mask=*a0
        move.l      PS.ptr(a6),a0             ; a0=&ptr
        move.l      (a0),a0                   ; a0=ptr
*
*       Set Up Block Counters
*
        move.l      PS.dst(a6),a1             ; a1=image
        move.l      PS.size_x(a6),d0          ; d0=size_x
        add.l       d0,d0                     ; in shorts
        move.l      d0,LS.x_linc(a6)          ; x_linc=1 row
        move.l      PS.size_y(a6),d1          ; d1=size_y
        muls.w      d0,d1                     ; d1*=d0 (area)
        add.l       a1,d1                     ; d1+=image
        move.l      d1,LS.y_lim(a6)           ; y_lim=d1
        move.l      d0,d2                     ; d2=d0 (1 row)
        add.l       d0,d0                     ; d0*=2 (2 rows)
        move.l      d0,d5                     ; y_blk=d0
        subq.l      #4,d5                     ; y_blk-=x_blk
        add.l       d0,d0                     ; d0*=2 (4 rows)
        move.l      d0,LS.y_inc0(a6)          ; y_inc0=d0
        add.l       d0,d0                     ; d0*=2 (8 rows)
        sub.l       d2,d0                     ; d0-=d2 (7 rows)
        move.l      d0,LS.y_inc1(a6)          ; y_inc1=d0

        move.l      PS.norms(a6),a2           ; GetNorm pointer
        move.l      (a2),d2                   ; read normal
        move.l      4(a2),d3                  ; read normal
        move.l      PS.lpfbits(a6),d4         ; read lpfbits
        move.l      LS.x_linc(a6),a4          ; read x_linc
        move.l      LS.y_inc0(a6),a5          ; read y_inc0

@y      move.l      a4,a3                     ; x_lim=x_linc
        add.l       a1,a3                     ; x_lim+=baddr
@x      UVSTILL0                              ; process UV block 0,0
        UVSTILL0                              ; process UV block 1,0
        add.l       a5,a1                     ; (2) addr[0]+=y_inc
        cmp.l       LS.y_lim(a6),a1           ; (2+) addr[0]-limit?
        bge.w       @last                     ; if half height
        sub.l       #16,a1                    ; pointer=blk(0,1)
        UVSTILL0                              ; process UV block 0,1
        UVSTILL0                              ; process UV block 1,1
@last   sub.l       a5,a1                     ; (2) addr[0]+=y_inc

        cmp.l       a3,a1                     ; (2+) addr[0]-limit?
        blt.w       @x                        ; (4) if less then loopX
        add.l       LS.y_inc1(a6),a1          ; (2+) addr[0]+=y_inc
        cmp.l       LS.y_lim(a6),a1           ; (2+) addr[0]-limit?
        blt.w       @y                        ; (4) if less then loopY
```

```
*
*   Save Bit Buffer
*
        move.l      PS.data(a6),a2          ; spare=&data
        move.l      d6,(a2)                 ; update data
        move.l      PS.bno(a6),a2           ; spare=&bno
        move.l      d7,(a2)                 ; update bno
        move.l      PS.ptr(a6),a2           ; spare=&ptr
        move.l      a0,(a2)                 ; update ptr
*
        movem.l     (a7)+,d4-d7/a3-a5       ; restore registers
        unlk        a6                      ; remove locals
        rts                                 ; return
*
        ENDFUNC
*------------------------------------------
Klics2D1Send    FUNC    EXPORT
*
*   Klics2D1Send(short *dst, long size_x, long size_y, short *norms, unsigned long
*
PS      RECORD      8
dst     DS.L        1
size_x  DS.L        1
size_y  DS.L        1
norms   DS.L        1
ptr     DS.L        1
data    DS.L        1
bno     DS.L        1
        ENDR
*
LS      RECORD      0,DECR
x_lim   DS.L        1               ; x counter termination     row_start+
x_linc  DS.L        1               ; x termination increment   1 row
y_inc0  DS.L        1               ; y counter increment       4 rows
y_inc1  DS.L        1               ; y counter increment       7 rows
y_lim   DS.L        1               ; y counter termination     area
LSize   EQU         *
        ENDR
*
*   d0/d1 - spare
*   d2 - step 0 (HH)
*   d3 - step 0
*   d4 - y_inc0
*   d5 - y_blk
*   d6 - data   (bit stream)
*   d7 - bno    (bit pointer)
*
*   a0 - ptr    (bit buffer)
*   a1 - baddr  (block address)
*   a2 - caddr  (coeff address)
*   a3 - x_lim
*   a4 - x_linc
*   a5 - y_lim
*
        link        a6,#LS.LSize            ; locals
        movem.l     d4-d7/a3-a5,-(a7)       ; store registers
*
*   Load Bit Buffer
*
        move.l      PS.data(a6),a0          ; a0=&data
        move.l      (a0),d6                 ; data=*a0
        move.l      PS.bno(a6),a0           ; a0=&mask
        move.l      (a0),d7                 ; mask=*a0
```

Engineering:KlicsCode:CompPict:KlicsDec2.a

```
        move.l      PS.ptr(a6),a0           ; a0=&ptr
        move.l      (a0),a0                 ; a0=ptr
*
*   Set Up Block Counters
*
        move.l      PS.dst(a6),a1           ; a1=image
        move.l      PS.size_x(a6),d0        ; d0=size_x
        add.l       d0,d0                   ; in shorts
        move.l      d0,LS.x_linc(a6)        ; x_linc=1 row
        move.l      PS.size_y(a6),d1        ; d1=size_y
        muls.w      d0,d1                   ; d1*=d0 (area)
        add.l       a1,d1                   ; d1+=image
        move.l      d1,LS.y_lim(a6)         ; y_lim=d1
        move.l      d0,d2                   ; d2=d0 (1 row)
        add.l       d0,d0                   ; d0*=2 (2 rows)
        move.l      d0,d5                   ; copy to d5
        subq.l      #4,d5                   ; subtract x_blk
        add.l       d0,d0                   ; d0*=2 (4 rows)
        move.l      d0,LS.y_inc0(a6)        ; y_inc0=d0
        add.l       d0,d0                   ; d0*=2 (8 rows)
        sub.l       d2,d0                   ; d0-=d2 (7 rows)
        move.l      d0,LS.y_inc1(a6)        ; y_inc1=d0

        move.l      PS.norms(a6),a2         ; GetNorm pointer
        move.l      (a2),d2                 ; read normal
        move.l      4(a2),d3                ; read normal
        move.l      LS.x_linc(a6),a4        ; read x_linc
        move.l      LS.y_inc0(a6),d4        ; read y_inc0
        move.l      LS.y_lim(a6),a5         ; read y_lim

@y      move.l      a4,a3                   ; x_lim=x_linc
        add.l       a1,a3                   ; x_lim+=baddr
@x      UVSENDO                             ; process UV block 0,0
        UVSENDO                             ; process UV block 1,0
        add.l       d4,a1                   ; (2) addr[0]+=y_inc
        cmp.l       a5,a1                   ; (2) addr[0]-limit?
        bge.w       @last                   ; if half height
        sub.l       #16,a1                  ; pointor=blk(0,1)
        UVSENDO                             ; process UV block 0,1
        UVSENDO                             ; process UV block 1,1
@last   sub.l       d4,a1                   ; (2) addr[0]+=y_inc

        cmp.l       a3,a1                   ; (2) addr[0]-limit?
        blt.w       @x                      ; (4) if less then loopX
        add.l       LS.y_inc1(a6),a1        ; (2+) addr[0]+=y_inc
        cmp.l       a5,a1                   ; (2) addr[0]-limit?
        blt.w       @y                      ; (4) if less then loopY
*
*   Save Bit Buffer
*
        move.l      PS.data(a6),a2          ; spare=&data
        move.l      d6,(a2)                 ; update data
        move.l      PS.bno(a6),a2           ; spare=&bno
        move.l      d7,(a2)                 ; update bno
        move.l      PS.ptr(a6),a2           ; spare=&ptr
        move.l      a0,(a2)                 ; update ptr
*
        movem.l     (a7)+,d4-d7/a3-a5       ; restore registers
        unlk        a6                      ; remove locals
        rts                                 ; return
*
        ENDFUNC
*--------------------------------
```

Engineering:KlicsCode:CompPict:KlicsDec2.a

```
KlicsD2Still   FUNC     EXPORT
*
*   Klics3D2Still(short *dst, long size_x, long size_y, long lpfbits, short *norms
*
PS       RECORD       8
dst      DS.L         1
size_x   DS.L         1
size_y   DS.L         1
lpfbits  DS.L         1
norms    DS.L         1
ptr      DS.L         1
data     DS.L         1
bno      DS.L         1
sub_tab  DS.L         1
         ENDR
*
LS       RECORD       0,DECR
y_blk0   DS.L         1                  ; y inter-block increment   2 rows - 4
y_blk1   DS.L         1                  ; y inter-block increment   4 rows - 8
x_inc    DS.L         1                  ; x counter increment       16
x_lim    DS.L         1                  ; x counter termination     row_start+
x_linc   DS.L         1                  ; x termination increment   1 row
y_inc    DS.L         1                  ; y counter increment       7 rows
y_lim    DS.L         1                  ; y counter termination     area
LSize    EQU          *
         ENDR
*
*   d0/d1 - spare
*   d2 - step 2HH
*   d3 - step 1
*   d4 - step 0/lpfbits
*   d5 - y_blk0,y_blk1
*   d6 - data    (bit stream)
*   d7 - bno     (bit pointer)
*
*   a0 - ptr     (bit buffer)
*   a1 - baddr   (block address)
*   a2 - caddr   (coeff address)
*   a3 - addrs   (tree addresses)
*   a4 - x_lim   (x counter termination)
*   a5 - lpfbits/step 0
*
        link        a6,#LS.LSize           ; locals
        movem.l     d4-d7/a3-a5,-(a7)      ; store registers
*
*   Load Bit Buffer
*
        move.l      PS.data(a6),a0         ; a0=&data
        move.l      (a0),d6                ; data=*a0
        move.l      PS.bno(a6),a0          ; a0=&mask
        move.l      (a0),d7                ; mask=*a0
        move.l      PS.ptr(a6),a0          ; a0=&ptr
        move.l      (a0),a0                ; a0=ptr
*
*   Set Up Block Counters
*
        move.l      PS.dst(a6),a1          ; a1=image
        move.l      PS.size_x(a6),d0       ; d0=size_x
        move.l      #16,LS.x_inc(a6)       ; save x_inc
        add.l       d0,d0                  ; in shorts
        move.l      d0,LS.x_linc(a6)       ; x_linc=1 row
        move.l      PS.size_y(a6),d1       ; d1=size_y
        muls.w      d0,d1                  ; d1*=d0 (area)
```

Engineering:KlicsCode:CompPict:KlicsDec2.a

```
        add.l       a1,d1                   ; d1+=image
        move.l      d1,LS.y_lim(a6)         ; y_lim=d1
        move.l      d0,d2                   ; d2=d0 (1 row)
        add.l       d0,d0                   ; d0*=2 (2 rows)
        move.l      d0,d5                   ; copy to d5
        subq.l      #4,d5                   ; y_blk: subtract x_blk
        move.l      d5,LS.y_blk0(a6)        ; save y_blk0
        add.l       d0,d2                   ; d2+=d0 (3 rows)
        add.l       d0,d0                   ; d0*=2  (4 rows)
        move.l      d0,d4                   ; copy to d5
        subq.l      #8,d4                   ; y_blk: subtract x_blk
        move.l      d4,LS.y_blk1(a6)        ; save y_blk1
        add.l       d2,d0                   ; d0+=d2 (7 rows)
        move.l      d0,LS.y_inc(a6)         ; y_inc=d0

        move.l      PS.norms(a6),a2         ; GetNorm pointer
        move.l      (a2),d2                 ; read normal
        move.l      4(a2),d3                ; read normal 1
        move.l      8(a2),a5                ; read normal 0
        move.l      PS.lpfbits(a6),d4       ; read lpfbits
        swap        d5                      ; y_blk=00XX
        move.l      LS.y_blk1(a6),d0        ; read y_blk1
        move.w      d0,d5                   ; d5=y_blk0/1
        move.l      PS.sub_tab(a6),a3       ; a3=addrs

@y      move.l      LS.x_linc(a6),a4        ; x_lim=x_linc
        add.l       a1,a4                   ; x_lim+=baddr
*
*   Low_Pass
*
@x      move.l      a1,a2                   ; caddr=baddr
        LPFSTILL    #8,d5,d2,d4
*
*   Sub-band gh
*
        bsr         DOSTILL2
        add.l       #20,a3
*
*   Sub-band hg
*
        bsr         DOSTILL2
        add.l       #20,a3
*
*   Sub-band gg
*
        bsr         DOSTILL2
        sub.l       #40,a3

        add.l       #16,a1                  ; (2) addr[0]+=x_inc
        cmp.l       a4,a1                   ; (2) addr[0]-limit?
        blt.w       @x                      ; (4) if less then loopX
        add.l       LS.y_inc(a6),a1         ; (2+) addr[0]+=y_inc
        cmp.l       LS.y_lim(a6),a1         ; (2+) addr[0]-limit?
        blt.w       @y                      ; (4) if less then loopY
*
*   Save Bit Buffer
*
@end    move.l      PS.data(a6),a2          ; spare=&data
        move.l      d6,(a2)                 ; update data
        move.l      PS.bno(a6),a2           ; spare=&bno
        move.l      d7,(a2)                 ; update bno
        move.l      PS.ptr(a6),a2           ; spare=&ptr
        move.l      a0,(a2)                 ; update ptr
```

```
*
        movem.l     (a7)+,d4-d7/a3-a5          ; restore registers
        unlk        a6                         ; remove locals
        rts                                    ; return
*
        ENDFUNC
*--------------------------------
Klics3D2Send    FUNC    EXPORT
*
*   Klics3D2Send(short *dst, long size_x, long size_y, short *norms, unsigned long
*
PS      RECORD      8
dst     DS.L        1
size_x  DS.L        1
size_y  DS.L        1
norms   DS.L        1
ptr     DS.L        1
data    DS.L        1
bno     DS.L        1
sub_tab DS.L        1
        ENDR
*
LS      RECORD      0,DECR
y_blk0  DS.L        1                          ; y inter-block increment   2 rows - 4
y_blk1  DS.L        1                          ; y inter-block increment   4 rows - 8
x_inc   DS.L        1                          ; x counter increment       16
x_lim   DS.L        1                          ; x counter termination     row_start+
x_linc  DS.L        1                          ; x termination increment   1 row
y_inc   DS.L        1                          ; y counter increment       7 rows
y_lim   DS.L        1                          ; y counter termination     area
LSize   EQU         *
        ENDR
*
*   d0 - spare
*   d1 - y_blk1
*   d2 - step 2HH
*   d3 - step 1
*   d4 - step 0
*   d5 - y_blk0
*   d6 - data   (bit stream)
*   d7 - bno    (bit pointer)
*
*   a0 - ptr    (bit buffer)
*   a1 - baddr  (block address)
*   a2 - caddr  (coeff address)
*   a3 - addrs  (tree addresses)
*   a4 - x_lim  (x counter termination)
*
        link        a6,#LS.LSize               ; locals
        movem.l     d4-d7/a3-a5,-(a7)          ; store registers
*
*   Load Bit Buffer
*
        move.l      PS.data(a6),a0             ; a0=&data
        move.l      (a0),d6                    ; data=*a0
        move.l      PS.bno(a6),a0              ; a0=&mask
        move.l      (a0),d7                    ; mask=*a0
        move.l      PS.ptr(a6),a0              ; a0=&ptr
        move.l      (a0),a0                    ; a0=ptr
*
*   Set Up Block Counters
*
        move.l      PS.dst(a6),a1              ; a1=image
```

```
        move.l      PS.size_x(a6),d0        ; d0=size_x
        move.l      #16,LS.x_inc(a6)        ; save x_inc
        add.l       d0,d0                   ; in shorts
        move.l      d0,LS.x_linc(a6)        ; x_linc=1 row
        move.l      PS.size_y(a6),d1        ; d1=size_y
        muls.w      d0,d1                   ; d1*=d0 (area)
        add.l       a1,d1                   ; d1+=image
        move.l      d1,LS.y_lim(a6)         ; y_lim=d1
        move.l      d0,d2                   ; d2=d0 (1 row)
        add.l       d0,d0                   ; d0*=2 (2 rows)
        move.l      d0,d5                   ; copy to d5
        subq.l      #4,d5                   ; y_blk: subtract x_blk
        move.l      d5,LS.y_blk0(a6)        ; save y_blk0
        add.l       d0,d2                   ; d2+=d0 (3 rows)
        add.l       d0,d0                   ; d0*=2  (4 rows)
        move.l      d0,d4                   ; copy to d5
        subq.l      #8,d4                   ; y_blk: subtract x_blk
        move.l      d4,LS.y_blk1(a6)        ; save y_blk1
        add.l       d2,d0                   ; d0+=d2 (7 rows)
        move.l      d0,LS.y_inc(a6)         ; y_inc=d0

        move.l      PS.norms(a6),a2         ; GetNorm pointer
        move.l      (a2),d2                 ; read normal
        move.l.     4(a2),d3                ; read normal 1
        move.l      8(a2),d4                ; read normal 0
        move.l      LS.y_blk1(a6),d1        ; read y_blk1
        move.l      PS.sub_tab(a6),a3       ; a3=addrs

@y      move.l      LS.x_linc(a6),a4        ; x_lim=x_linc
        add.l       a1,a4                   ; x_lim+=baddr
*
*   Low_Pass
*
@x      buf_rinc    a0,d6,d7                ; BUF_INC
        buf_get     d6,d7                   ; BUF_GET
        beq.w       @subs                   ; if 0 then process subbands
        move.l      a1,a2                   ; caddr=baddr
        SEND        #8,d1,d2
*
*   Sub-band gh
*
@subs   bsr         DOSEND2
        add.l       #20,a3
*
*   Sub-band hg
*
        bsr         DOSEND2
        add.l       #20,a3
*
*   Sub-band gg
*
        bsr         DOSEND2
        sub.l       #40,a3

        add.l       #16,a1                  ; (2) addr[0]+=x_inc
        cmp.l       a4,a1                   ; (2) addr[0]-limit?
        blt.w       @x                      ; (4) if less then loopX
        add.l       LS.y_inc(a6),a1         ; (2+) addr[0]+=y_inc
        cmp.l       LS.y_lim(a6),a1         ; (2+) addr[0]-limit?
        blt.w       @y                      ; (4) if less then loopY
*
*   Save Bit Buffer
*
```

```
@end    move.l    PS.data(a6),a2          ; spare=&data
        move.l    d6,(a2)                 ; update data
        move.l    PS.bno(a6),a2           ; spare=&bno
        move.l    d7,(a2)                 ; update bno
        move.l    PS.ptr(a6),a2           ; spare=&ptr
        move.l    a0,(a2)                 ; update ptr
*
        movem.l   (a7)+,d4-d7/a3-a5       ; restore registers
        unlk      a6                      ; remove locals
        rts                               ; return
*
        ENDFUNC
*---------------------------------
        END
```

Engineering:KlicsCode:CompPict:KlicsDec.c

```
/******************************************
 *
 * © Copyright 1993 KLICS Limited
 * All rights reserved.
 *
 * Written by: Adrian Lewis
 *
 *******************************/
/*
 * Importing raw Klics binary files
 *
 * Stand-alone version
 */

#include    "Bits3.h"
#include    "Klics.h"
#include    "KlicsHeader.h"

typedef char    Boolean;

/* If bool true the negate value */
#define negif(bool,value)   ((bool)?-(value):(value))

extern  void    HaarBackward();
extern  void    Daub4Backward(short *data,int size[2],int oct_src);
extern  void    TestTopBackward(short *data,int size[2],int oct_src);
extern  void    TestBackward(short *data,int size[2],int oct_src);
extern  void    KLICSDCHANNEL(short *dst, long octs, long size_x, long size_y, lon
/* Use the bit level file macros (Bits2.h) */
/* buf_use; */

/* Huffman decode a block */
#define HuffDecLev(lev,buf) \
    lev[0]=HuffDecode(buf); \
    lev[1]=HuffDecode(buf); \
    lev[2]=HuffDecode(buf); \
    lev[3]=HuffDecode(buf);

/* Fixed length decode block of integers */
#define IntDecLev(lev,lpf_bits,buf) \
    lev[0]=IntDecode(lpf_bits,buf); \
    lev[1]=IntDecode(lpf_bits,buf); \
    lev[2]=IntDecode(lpf_bits,buf); \
    lev[3]=IntDecode(lpf_bits,buf);

/* Reverse quantize difference block */
#define RevQntDelta(new,old,lev,shift) \
    new[0]=old[0]+(lev[0]<<shift)+(lev[0]!=0?negif(lev[0]<0,(1<<shift)-1>>1):0); \
    new[1]=old[1]+(lev[1]<<shift)+(lev[1]!=0?negif(lev[1]<0,(1<<shift)-1>>1):0); \
    new[2]=old[2]+(lev[2]<<shift)+(lev[2]!=0?negif(lev[2]<0,(1<<shift)-1>>1):0); \
    new[3]=old[3]+(lev[3]<<shift)+(lev[3]!=0?negif(lev[3]<0,(1<<shift)-1>>1):0);

/* Reverse quantize block */
#define RevQnt(new,lev,shift) \
    new[0]=(lev[0]<<shift)+(lev[0]!=0?negif(lev[0]<0,(1<<shift)-1>>1):0); \
    new[1]=(lev[1]<<shift)+(lev[1]!=0?negif(lev[1]<0,(1<<shift)-1>>1):0); \
    new[2]=(lev[2]<<shift)+(lev[2]!=0?negif(lev[2]<0,(1<<shift)-1>>1):0); \
    new[3]=(lev[3]<<shift)+(lev[3]!=0?negif(lev[3]<0,(1<<shift)-1>>1):0);

#define RevQntLPF(new,lev,shift) \
    new[0]=(lev[0]<<shift)+((1<<shift)-1>>1); \
    new[1]=(lev[1]<<shift)+((1<<shift)-1>>1); \
    new[2]=(lev[2]<<shift)+((1<<shift)-1>>1); \
```

```
    new[3]=Plev[3]<<shift)+((1<<shift.-1>>1);

/* Read a difference block and update memory */
#define DoXferDelta(addr,old,new,lev,dst,shift,mode,oct,nmode,buf) \
    HuffDecLev(lev,buf); \
    RevQntDelta(new,old,lev,shift) \
    PutData(addr,new,dst); \
    mode[oct]=oct==0?M_STOP:nmode;

/* Read a block and update memory */
#define DoXfer(addr,new,lev,dst,shift,mode,oct,nmode,buf) \
    HuffDecLev(lev,buf); \
    RevQnt(new,lev,shift) \
    PutData(addr,new,dst); \
    mode[oct]=oct==0?M_STOP:nmode;

/*  Function Name:  IntDecode
 *  Description:    Read a integer from bit file
 *  Arguments:  bits - bits/integer now signed
 *  Returns:    integer value
 */

short   IntDecode(short bits,Buf buf)
{
    int     i, lev=0, mask=1;
    Boolean sign;

    /* Hardware compatatble version */
    buf_rinc(buf);
    sign=buf_get(buf);
    for(i=0;i<bits-1;i++) {
        buf_rinc(buf);
        if (buf_get(buf)) lev |= mask;
        mask <<= 1;
    }
    if (sign) lev= -lev;
    return(lev);
}

/*  Function Name:  HuffDecode
 *  Description:    Read a Huffman coded integer from bit file
 *  Returns:    integer value
 */

short   HuffDecode(Buf buf)

{
    short   lev=0, i;
    Boolean neg;

    /* Hardware compatatble version */
    buf_rinc(buf);
    if (buf_get(buf)) {
        buf_rinc(buf);
        neg=buf_get(buf);
        do {
            buf_rinc(buf);
            lev++;
        } while (lev<7 && !(buf_get(buf)));
        if (!(buf_get(buf))) {
            for(lev=0,i=0;i<7;i++) {
                lev<<=1;
                buf_rinc(buf);
```

```
                if (buf_get(buf)) lev++;
            }
            lev+=8;
        }
        if (neg) lev= -lev;
    }
    return(lev);
}

/*  Function Name:  KlicsDChannel
 *  Description:    Decode a channel of image
 *  Arguments:  dst - destination memory (and old for videos)
 *              octs, size - octaves of decomposition and image dimensions
 *              normals - HVS weighted normals
 *              lpf_bits - no of bits for LPF integer (image coding only)
 */

void    KlicsDecY(short *dst, int octs, int size[2], KlicsFrameHeader *frmh,
    KlicsSeqHeader *seqh, Buf buf)
{
    int     oct, mask, x, y, sub, step=2<<octs, blk[4], mode[4], base_mode=(frmh->
    Blk     addr, new, old, lev;

    for(y=0;y<size[1];y+=step)
    for(x=0;x<size[0];x+=step)
    for(sub=0;sub<4;sub++) {
    mode[oct=octs-1]=base_mode;
    if (sub==0) mode[oct=octs-1] |= M_LPF;
    mask=2<<oct;
    do {
        GetAddr(addr,x,y,sub,oct,size,mask);
        switch(mode[oct]) {
        case M_VOID:
            GetData(addr,old,dst);
            if (BlkZero(old)) mode[oct]=M_STOP;
            else { DoZero(addr,dst,mode,oct); }
            break;
        case M_SEND|M_STILL:
            buf_rinc(buf);
            if (buf_get(buf)) {
                buf_rinc(buf);
                if (buf_get(buf)) {
                    DoZero(addr,dst,mode,oct);
                } else {
                    DoXfer(addr,new,lev,dst,frmh->quantizer[octs-oct],mode,oct,M_S
                }
            } else
                mode[oct]=M_STOP;
            break;
        case M_SEND:
            buf_rinc(buf);
            if (buf_get(buf)) {
                buf_rinc(buf);
                if (buf_get(buf)) {
                    buf_rinc(buf);
                    if (buf_get(buf)) {
                        GetData(addr,old,dst);
                        DoXferDelta(addr,old,new,lev,dst,frmh->quantizer[octs-oct]
                    } else {
                        DoZero(addr,dst,mode,oct);
                    }
                } else {
                    DoXfer(addr,new,lev,dst,frmh->quantizer[octs-oct],mode,oct,M_S
```

```
            }
        } else
            mode[oct]=M_STOP;
        break;
    case M_STILL:
        buf_rinc(buf);
        if (buf_get(buf)) { DoXfer(addr,new,lev,dst,frmh->quantizer[octs-oct],
        else mode[oct]=M_STOP;
        break;
    case M_LPF|M_STILL:
        IntDecLev(lev,seqh->precision-frmh->quantizer[0],buf);
        RevQntLPF(new,lev,frmh->quantizer[0]);
        PutData(addr,new,dst);
        mode[oct]=M_QUIT;
        break;
    case M_LPF|M_SEND:
        buf_rinc(buf);
        if (buf_get(buf)) {
            GetData(addr,old,dst);
            HuffDecLev(lev,buf);
            RevQntDelta(new,old,lev,frmh->quantizer[0]);
            PutData(addr,new,dst);
        }
        mode[oct]=M_QUIT;
        break;
    }
    switch(mode[oct]) {
    case M_STOP:
        StopCounters(mode,oct,mask,blk,x,y,octs);
        break;
    case M_QUIT:
        break;
    default:
        DownCounters(mode,oct,mask,blk);
        break;
    }
  } while (mode[oct]!=M_QUIT);
  }
}

void    KlicsDecUV(short *dst, int octs, int size[2], KlicsFrameHeader *frmh,
    KlicsSeqHeader *seqh, Buf buf)
{
    int     oct, mask, x, y, X, Y, sub, step=4<<octs, blk[4], mode[4], base_mode=(
    Blk     addr, new, old, lev;

    for(Y=0;Y<size[1];Y+=step)
    for(X=0;X<size[0];X+=step)
    for(y=Y;y<size[1] && y<Y+step;y+=step>>1)
    for(x=X;x<size[0] && x<X+step;x+=step>>1)
    for(sub=0;sub<4;sub++) {
    mode[oct=octs-1]=base_mode;
    if (sub==0) mode[oct=octs-1] |= M_LPF;
    mask=2<<oct;
    do {
        GetAddr(addr,x,y,sub,oct,size,mask);
        switch(mode[oct]) {
        case M_VOID:
            GetData(addr,old,dst);
            if (BlkZero(old)) mode[oct]=M_STOP;
            else { DoZero(addr,dst,mode,oct); }
            break;
        case M_SEND|M_STILL:
```

```
        buf_rinc(buf);
        if (buf_get(buf)) {
            buf_rinc(buf);
            if (buf_get(buf)) {
                DoZero(addr,dst,mode,oct);
            } else {
                DoXfer(addr,new,lev,dst,frmh->quantizer[octs-oct],mode,oct,M_S
            }
        } else
            mode[oct]=M_STOP;
        break;
    case M_SEND:
        buf_rinc(buf);
        if (buf_get(buf)) {
            buf_rinc(buf);
            if (buf_get(buf)) {
                buf_rinc(buf);
                if (buf_get(buf)) {
                    GetData(addr,old,dst);
                    DoXferDelta(addr,old,new,lev,dst,frmh->quantizer[octs-oct]
                } else {
                    DoZero(addr,dst,mode,oct);
                }
            } else {
                DoXfer(addr,new,lev,dst,frmh->quantizer[octs-oct],mode,oct,M_S
            }
        } else
            mode[oct]=M_STOP;
        break;
    case M_STILL:
        buf_rinc(buf);
        if (buf_get(buf)) { DoXfer(addr,new,lev,dst,frmh->quantizer[octs-oct],
        else mode[oct]=M_STOP;
        break;
    case M_LPF|M_STILL:
        IntDecLev(lev,seqh->precision-frmh->quantizer[0],buf);
        RevQntLPF(new,lev,frmh->quantizer[0]);
        PutData(addr,new,dst);
        mode[oct]=M_QUIT;
        break;
    case M_LPF|M_SEND:
        buf_rinc(buf);
        if (buf_get(buf)) {
            GetData(addr,old,dst);
            HuffDecLev(lev,buf);
            RevQntDelta(new,old,lev,frmh->quantizer[0]);
            PutData(addr,new,dst);
        }
        mode[oct]=M_QUIT;
        break;
    }
    switch(mode[oct]) {
    case M_STOP:
        StopCounters(mode,oct,mask,blk,x,y,octs);
        break;
    case M_QUIT:
        break;
    default:
        DownCounters(mode,oct,mask,blk);
        break;
    }
} while (mode[oct]!=M_QUIT);
}
```

```
)

/*  Function Name:  KlicsDecode
 *  Description:    Decode a frame to YUV (de)transformed image
 *  Arguments:  src - destination result
 *              dst - transformed destination memory (and old for videos)
 *  Returns:    whether this frame was skipped
 */

extern  void    KLCOPY(short *dst, short *src, long area);
extern  void    KLHALF(short *dst, short *src, long size_0, long size_1);
extern  void    KLICS3D2SEND(short *dst, long size_x, long size_y, short norms[4][
extern  void    KLICS2D1STILL(short *dst, long size_x, long size_y, long lpfbits,
extern  void    KLICS3D2STILL(short *dst, long size_x, long size_y, long lpfbits,
extern  void    KLICS2D1SEND(short *dst, long size_x, long size_y, short norms[4][

#define flag_tree   0x1
#define flag_wave   0x2

void    KlicsDecode(short *src[3], short *dst[3], KlicsSeqHeader *seqh,KlicsFrameH
{
    long    channel,i;
    short   norms[4][2];
    unsigned long sync1, sync2;

    for(i=0;i<4;i++) {
        norms[i][0]=(1<<frmh->quantizer[i]-1)-1;
        norms[i][1]=frmh->quantizer[i];
    }
    buf_rinit(buf);
    if (0!=(flags&flag_tree)) {
        sync1=GetTimerValue(&sync1);
        for(channel=0;channel<seqh->channels;channel++) {
            int     size[2]={seqh->sequence_size[0]>>(channel==0?0:seqh->sub_sampl
                        seqh->sequence_size[1]>>(channel==0?0:seqh->sub_sample[1])
                    tree_size[2]={size[0]>>scale[0],size[1]>>scale[0]},
                    octs=seqh->octaves[channel==0?0:1];

#ifdef HQ
            if (0!=(frmh->flags&KFH_INTRA))
                KLZERO(dst[channel],tree_size[0]*tree_size[1]);
/*          KLICSDCHANNEL(dst[channel],octs-1,tree_size[0],tree_size[1],(long)(seq
            if (channel==0) KlicsDecY(dst[channel],octs,tree_size,frmh,seqh,buf);
            else KlicsDecUV(dst[channel],octs,tree_size,frmh,seqh,buf);
#else
            long    sub_tab[15]={4,2,10,2+8*tree_size[0],10+8*tree_size[0],
                        4*tree_size[0],2*tree_size[0],8+2*tree_size[0],10*tree_siz
                        4+4*tree_size[0],2+2*tree_size[0],10+2*tree_size[0],2+10*t

            if (0!=(frmh->flags&KFH_INTRA)) {
                KLZERO(dst[channel],tree_size[0]*tree_size[1]);
                if (octs==3)
                    KLICS3D2STILL(dst[channel],tree_size[0],tree_size[1],(long)(se
                else
                    KLICS2D1STILL(dst[channel],tree_size[0],tree_size[1],(long)(se
            } else
                if (octs==3)
                    KLICS3D2SEND(dst[channel],tree_size[0],tree_size[1],&norms,&bu
                else
                    KLICS2D1SEND(dst[channel],tree_size[0],tree_size[1],&norms,&bu
#endif
        }
        sync2=GetTimerValue(&sync2);
```

```
            *tree=sync2-sync1;
        }
        if (0!=(flags&flag_wave)) {
            sync1=GetTimerValue(&sync1);
            for(channel=0;channel<seqh->channels;channel++) {
                int     size[2]={seqh->sequence_size[0]>>(channel==0?0:seqh->sub_sampl
                            seqh->sequence_size[1]>>(channel==0?0:seqh->sub_sample[1])
                        wave_size[2]={size[0]>>scale[1],size[1]>>scale[1]},
                        octs=seqh->octaves[channel==0?0:1];

                switch(seqh->wavelet) {
                case WT_Haar:
                    if (scale[1]>scale[0])
                        KLHALF(dst[channel],src[channel],wave_size[0],wave_size[1]);
                    else
                        KLCOPY(dst[channel],src[channel],wave_size[0]*wave_size[1]);
                    HaarBackward(src[channel],wave_size,octs-scale[1]);
                    break;
                case WT_Daub4:
                    if (scale[0]==0) {
                        if (scale[1]>scale[0])
                            KLHALF(dst[channel],src[channel],wave_size[0],wave_size[1]
                        else
                            KLCOPY(dst[channel],src[channel],wave_size[0]*wave_size[1]
                        Daub4Backward(src[channel],wave_size,octs-scale[1]);
                    } else
                        if (channel==0) {
                            KLCOPY(dst[channel],src[channel],wave_size[0]*wave_size[1]
                            Backward3511(src[channel],wave_size,octs-scale[1]);
                        } else
                            TOPBWD(dst[channel],src[channel],wave_size[0],wave_size[1]
                    break;
                }
            }
            sync2=GetTimerValue(&sync2);
            *wave=sync2-sync1;
        }
    }
```

Engineering:KlicsCode:CompPict:KlicsCodec.c

```
/*******************************
 *
 *  © Copyright 1993 KLICS Limited
 *  All rights reserved.
 *
 *  Written by: Adrian Lewis
 *
 ********************************/
/*
 *  Klics Codec
 */

#include "ImageCodec.h"
#include <FixMath.h>
#include <Errors.h>
#include <Packages.h>

#ifdef PERFORMANCE
    #include <Perf.h>
    extern  TP2PerfGlobals  ThePGlobals;
#endif

#ifdef DEBUG
    #define DebugMsg(val)    DebugStr(val)
#else
    #define DebugMsg(val)
#endif


#define WT_Haar  0
#define WT_Daub4 1

#define None    0
#define Use8    1
#define Use16   2
#define Use32   3
#define UseF32  4

/* Version information */

#define KLICS_CODEC_REV          1
#define codecInterfaceVersion    1   /* high word returned in component GetVersion

#define klicsCodecFormatName     "Klics"
#define klicsCodecFormatType     'klic'

pascal ComponentResult
KlicsCodec(ComponentParameters *params,char **storage);

pascal ComponentResult
KLOpenCodec(ComponentInstance self);

pascal ComponentResult
KLCloseCodec(Handle storage,ComponentInstance self);

pascal ComponentResult
KLCanDoSelector(short selector);

pascal ComponentResult
KLGetVersion();

pascal ComponentResult
KLGetCodecInfo(Handle storage,CodecInfo *info);
```

```
pascal ComponentResult
KLGetMaxCompressionSize(Handle storage,PixMapHandle src,const Rect *srcRect,short
    CodecQ quality,long *size);

pascal ComponentResult
KLGetCompressedImageSize(Handle storage,ImageDescriptionHandle desc,Ptr data,long
    DataProcRecordPtr dataProc,long *size);

pascal ComponentResult
KLPreCompress(Handle storage,register CodecCompressParams *p);

pascal long
KLPreDecompress(Handle storage,register CodecDecompressParams *p);

pascal long
KLBandDecompress(Handle storage,register CodecDecompressParams *p);

pascal long
KLBandCompress(Handle storage,register CodecCompressParams *p);

pascal ComponentResult
KLGetCompressionTime(Handle storage,PixMapHandle src,const Rect *srcRect,short dep
        CodecQ *spatialQuality,CodecQ *temporalQuality,unsigned long *time);

/*  Function:   KlicsCodec
 *  Description:KlicsCodec main despatcher
 */

#ifdef DECODER
pascal ComponentResult
KlicsDecoder(ComponentParameters *params,char **storage)
#else
#ifdef ENCODER
pascal ComponentResult
KlicsEncoder(ComponentParameters *params,char **storage)
#else
pascal ComponentResult
KlicsCodec(ComponentParameters *params,char **storage)
#endif
#endif
{
    OSErr   err;

    switch ( params->what ) {
    case kComponentOpenSelect:
        err=CallComponentFunction(params,(ComponentFunction) KLOpenCodec); break;

    case    kComponentCloseSelect:
        err=CallComponentFunctionWithStorage(storage,params,(ComponentFunction)KLC

    case    kComponentCanDoSelect:
        err=CallComponentFunction(params,(ComponentFunction)KLCanDoSelector); brea

    case kComponentVersionSelect :
        err=CallComponentFunction(params,(ComponentFunction)KLGetVersion); break;

#ifdef DECODER
    case codecPreCompress:
    case codecBandCompress:
        err=codecUnimpErr; break;
#else
    case codecPreCompress:
```

Engineering:KlicsCode:CompPict:KlicsCodec.c

```
        err=CallComponentFunctionWithStorage(storage.params.(ComponentFunction)KLP

    case codecBandCompress:
        err=CallComponentFunctionWithStorage(storage.params.(ComponentFunction)KLB
#endif
#ifdef ENCODER
    case codecPreDecompress:
    case codecBandDecompress:
        err=codecUnimpErr; break;
#else
    case codecPreDecompress:
        err=CallComponentFunctionWithStorage(storage,params,(ComponentFunction)KLP

    case codecBandDecompress:
        err=CallComponentFunctionWithStorage(storage,params,(ComponentFunction)KLB
#endif
    case codecCDSequenceBusy:
        err=0; break;                   /* our codec is never asynchronously busy

    case codecGetCodecInfo:
        err=CallComponentFunctionWithStorage(storage,params,(ComponentFunction)KLG

    case codecGetCompressedImageSize:
        err=CallComponentFunctionWithStorage(storage,params,(ComponentFunction)KLG

    case codecGetMaxCompressionSize:
        err=CallComponentFunctionWithStorage(storage,params,(ComponentFunction)KLG

    case codecGetCompressionTime:
        err=CallComponentFunctionWithStorage(storage,params,(ComponentFunction)KLG

    case codecGetSimilarity:
        err=codecUnimpErr; break;

    case codecTrimImage:
        err=codecUnimpErr; break;

    default:
        err=paramErr; break;
    }
    if (err!=noErr)
        DebugMsg("\pCodec Error");
    return(err);
}

#include <Memory.h>
#include <Resources.h>
#include <OSUtils.h>
#include <SysEqu.h>

#include <StdIO.h>
#include <Time.h>

#include <Strings.h>
#include <String.h>
#include "Bits3.h"
#include "KlicsHeader.h"
#include "KlicsEncode.h"

void    DebugString(char *string)
{
    DebugStr(string);
}
```

```
extern  short   gResRef;

typedef struct {
    CodecInfo   **info;
    Ptr     tab[4];
    short   use[4];
} SharedGlobals;

typedef struct {
    KlicsERec   kle;                    /* Encoding parameters */
    short   *src[3];                    /* YUV Frame buffer */
    short   *dst[3];                    /* YUV Frame buffer */
    Ptr     pixmap;                     /* Encoded pixmap data */
    long    size;                       /* Size of Previous Frame Buffer */
    long    using;                      /* Which lookup table are we using for colour
    long    scale[3];                   /* Tree, Wave, Out scales 0=Original, -1=Doubl
    unsigned long prev_frame;           /* Previous frame number */
    unsigned long real_frame;           /* Previous real frame (no skips) */
    unsigned long dpy_frame;            /* Previous displayed frame */
    unsigned long run_frame;            /* First frame in play sequence */
    unsigned long sys_time;             /* System overhead for previous frame */
    unsigned long tree_time;            /* Typical tree decode time (not skip) */
    unsigned long wave_time;            /* Typical wavelet transform time */
    unsigned long dpy_time;             /* Typical display time */
    unsigned long run_time;             /* Time of first run frame */
    unsigned long key_time;             /* Time at last key frame */
    unsigned long sync_time;            /* Sync time */
    Boolean out[15];                    /* Displayed? */
    SharedGlobals   *sharedGlob;
} Globals;

/* Scaling scenarios: Tree Wave Out
 *
 *  1  1  0: Internal calculations are Quarter size, output Original size (interpo
 *  1  1  1: Internal calculations are Quarter size, output Quarter size
 *  0  1  1: Internal calculations are Original size, output Quarter size
 *  0  0  0: Internal calculations are Original size, output Original size
 *  0  0 -1: Internal calculations are Original size, output Double size
 */

void    KLDeallocate(Globals **glob);

/* Klics Function Definitions */

extern  int     KlicsEncode(short *src[3], short *dst[3], KlicsE kle);
extern  Boolean KlicsDecode(short *src[3], short *dst[3], KlicsSeqHeader *seqh,Kli
    long mode, long scale[3], unsigned long *tree, unsigned long *wave);

/*****************************************
 *
 *  Memory allocation/deallocation routines
 *
 *****************************************/

OSErr
MemoryError()
{
    OSErr   theErr;

#ifdef DEBUG
    if (0!=(theErr=MemError()))
        DebugStr("\pMemoryError");
```

```
#endif
    return(theErr);
}

OSErr
FreePtr(Ptr *ptr)
{
    OSErr   theErr=0;

    if (*ptr!=nil) {
        DisposePtr(*ptr);
        *ptr=nil;
        theErr=MemoryError();
    }
    return(theErr);

}

#define FreePointer(handle,err) \
    if (noErr!=(err=FreePtr((Ptr*)(&handle)))) return(err)

extern  OSErr   Colour8(Ptr *);
extern  OSErr   Colour16(Ptr *);
extern  OSErr   UV32Table(Ptr *);
extern  OSErr   RGBTable(Ptr *);

OSErr
KLGetTab(Globals **glob,long new)
{
    OSErr   theErr=0;
    SharedGlobals   *sGlob=(*glob)->sharedGlob;
    long    old=(*glob)->using;

    if (old!=new) {
        if (old!=None) {
            sGlob->use[old-1]--;
            if (sGlob->use[old-1]==0) {
                FreePointer(sGlob->tab[old-1],theErr);
            }
        }

        if (new!=None) {
            if (sGlob->use[new-1]==0)
                switch(new) {
#ifndef ENCODER
                case Use8:
                    if (noErr!=(theErr=Colour8(&sGlob->tab[new-1])))
                        return(theErr);
                    break;
                case Use16:
                    if (noErr!=(theErr=Colour16(&sGlob->tab[new-1])))
                        return(theErr);
                    break;
                case Use32:
                    if (noErr!=(theErr=UV32Table(&sGlob->tab[new-1])))
                        return(theErr);
                    break;
#endif
#ifndef DECODER
                case UseF32:
                    if (noErr!=(theErr=RGBTable(&sGlob->tab[new-1])))
                        return(theErr);
                    break;
```

```
#endif
                }
            (*glob)->using=new;
            sGlob->use[new-1]++;
        }
    }
    return(theErr);
}


OSErr
KLFree(Globals **glob)
{
    OSErr   theErr=0;

    FreePointer((*glob)->src[0],theErr);
    FreePointer((*glob)->dst[0],theErr);
    FreePointer((*glob)->pixmap,theErr);
    (*glob)->size=0;
    return(theErr);
}

#define NewPointer(ptr,type,size) \
    saveZone=GetZone(); \
    SetZone(SystemZone()); \
    if (nil==(ptr=(type)NewPtr(size))) { \
        SetZone(ApplicZone()); \
        if (nil==(ptr=(type)NewPtr(size))) { \
            SetZone(saveZone); \
            return(MemoryError()); \
        } \
    } \
    SetZone(saveZone);


ComponentResult
KLMalloc(Globals **glob, short height, short width, long pixelSize)
{
    long    ysize,uvsize;
    THz     saveZone;

    ysize= (long)height * (long)width * (long)sizeof(short);
    uvsize = ysize>>2;

    if ((*glob)->size != ysize) {
        KLFree(glob);
        (*glob)->size = ysize;
        (*glob)->prev_frame=-1; /* frame doesn't contain valid data */

        /* Keep Src and Dst separate because of their large sizes */

        ysize=(long)height * (long)width * (long)sizeof(short) >> 2*(*glob)->scale
        uvsize = ysize>>2;
        NewPointer((*glob)->src[0],short *,ysize+uvsize+uvsize+16);
        (*glob)->src[1] = (short *)(((long)(*glob)->src[0] + ysize + 3L) & 0xFFFFF
        (*glob)->src[2] = (short *)(((long)(*glob)->src[1] + uvsize + 3L) & 0xFFFF

        ysize=(long)height * (long)width * (long)sizeof(short) >> 2*(*glob)->scale
        uvsize = ysize>>2;
        NewPointer((*glob)->dst[0],short *,ysize+uvsize+uvsize+16);
        (*glob)->dst[1] = (short *)(((long)(*glob)->dst[0] + ysize + 3L) & 0xFFFFF
        (*glob)->dst[2] = (short *)(((long)(*glob)->dst[1] + uvsize + 3L) & 0xFFFF
```

```
        NewPointer((*glob)->pixmap.Ptr.pixelSize/8*height*width<<1);
    }
    return(noErr);
}

OSErr
ResourceError()
{
    OSErr   theErr;

#ifdef DEBUG
    if (0!=(theErr=ResError()))
        DebugStr("\pResourceError");
#endif
    return(theErr);
}

#ifdef COMPONENT
    #define ResErr(resfile,err) \
        if (0!=(err=ResourceError())) { \
            if (resfile!=0) CloseComponentResFile(resfile); \
            return(err); \
        }
#else
    #define ResErr(resfile,err) \
        if (0!=(err=ResourceError())) { \
            return(err); \
        }
#endif

ComponentResult
KLOpenInfoRes(ComponentInstance self, Handle *info)
{
#pragma unused(self)
    short   resFile=0;
    OSErr   theErr=noErr;

    if (*info) {
        DisposHandle(*info);
        *info=nil;
    }
#ifdef COMPONENT
    resFile=OpenComponentResFile((Component)self);
    ResErr(resFile,theErr);
#else
    UseResFile(gResRef);
#endif
    *info=Get1Resource(codecInfoResourceType,128);
    *info=Get1Resource(codecInfoResourceType,129);
    ResErr(resFile,theErr);
    LoadResource(*info);
    ResErr(resFile,theErr);
    DetachResource(*info);
#ifdef COMPONENT
    CloseComponentResFile(resFile);
#endif
    return(theErr);
}

pascal ComponentResult
KLOpenCodec(ComponentInstance self)
{
    Globals         **glob;
```

```
    SharedGlobals    *sGlob;
    THz              saveZone;
    Boolean          inAppHeap;
    ComponentResult  result = noErr;
    short    resFile=CurResFile();

    DebugMsg("\pOpen Codec - begin");
    if ( (glob = (Globals **)NewHandleClear(sizeof(Globals))) == nil )  {
        return(MemoryError());
    } else HNoPurge((Handle)glob);
    SetComponentInstanceStorage(self,(Handle)glob);

    saveZone = GetZone();
    inAppHeap = ( GetComponentInstanceA5(self) != 0 );
    if ( !inAppHeap )
        SetZone(SystemZone());
    if ( (sGlob=(SharedGlobals*)GetComponentRefcon((Component)self)) == nil  ) {
        if ( (sGlob = (SharedGlobals*)NewPtrClear(sizeof(SharedGlobals))) == nil )
            result=MemoryError();
            goto obail;
        }
        SetComponentRefcon((Component)self,(long)sGlob);
    }

    (*glob)->sharedGlob = sGlob;    // keep this around where it's easy to get at

    if ( sGlob->info == nil || *(Handle)sGlob->info == nil  )  {
        result=KLOpenInfoRes(self,&(Handle)(sGlob->info));
        HNoPurge((Handle)sGlob->info);
    }

obail:

    SetZone(saveZone);
    if ( result != noErr && sGlob != nil ) {
        if ( sGlob->info )
            DisposHandle((Handle)sGlob->info);
        DisposPtr((Ptr)sGlob);
        SetComponentRefcon((Component)self,(long)nil);
    }
    (*glob)->size=0;
    DebugMsg("\pOpen Codec - end");
    return(result);
}

pascal ComponentResult
KLCloseCodec(Handle storage,ComponentInstance self)
{
    SharedGlobals    *sGlob;
    Globals          **glob = (Globals **)storage;

    DebugMsg("\pClose Codec - begin");
    HLock(storage);
    if ( glob ) {
        KLFree(glob);
        KLGetTab(glob,None);
        if (CountComponentInstances((Component)self) == 1) {
            if ( (sGlob=(SharedGlobals*)(*glob)->sharedGlob) != nil  ) {
                if ( sGlob->info )
                    HPurge((Handle)sGlob->info);
            }
        }
        DisposHandle((Handle)glob);
```

```
        height = 120;
        }
    if (time)
        *time = (width * height * 1L);

    if (spatialQuality && *spatialQuality==codecLosslessQuality)
        *spatialQuality = codecMaxQuality;

    if (temporalQuality && *temporalQuality==codecLosslessQuality)
        *temporalQuality = codecMaxQuality;

    return(noErr);
}

/*
 * Extends dimensions to make a multiples of 32x16
 */

#define KLExtendWidth(dim)  31-(dim-1&31)
#define KLExtendHeight(dim) 15-(dim-1&15)

pascal ComponentResult
KLGetMaxCompressionSize(Handle storage,PixMapHandle src,const Rect *srcRect,short
    CodecQ quality,long *size)
{
#pragma unused(storage,src,depth,quality)
    short width = srcRect->right - srcRect->left;
    short height = srcRect->bottom - srcRect->top;

    /*  test by just doing RGB storage */

    *size = 3 * (width+KLExtendWidth(width)) * (height+KLExtendHeight(height));
    return(noErr);
}

pascal ComponentResult
KLGetCompressedImageSize(Handle storage,ImageDescriptionHandle desc,Ptr data,long
    DataProcRecordPtr dataProc,long *size)
{
#pragma unused(storage,dataSize,dataProc,desc)
    short   frmh_size;
    long    data_size;

    if ( size == nil ) {
        return(paramErr);
    }
    frmh_size=((KlicsHeader *)data)->description_length;
    data_size=((KlicsFrameHeader *)data)->length;
    *size=(long)frmh_size+data_size;
    return(noErr);
}

void    KLSetup(Boolean still, short width, short height, CodecQ space, CodecQ tem
{
    kle->seqh.head.description_length=sizeof(KlicsSeqHeader);
    kle->seqh.head.version_number[0]=0;
    kle->seqh.head.version_number[1]=1;
    kle->seqh.sequence_size[0]=width;
    kle->seqh.sequence_size[1]=height;
    kle->seqh.sequence_size[2]=0;
    kle->seqh.sub_sample[0]=1;
    kle->seqh.sub_sample[1]=1;
    kle->seqh.wavelet=WT_Daub4;
```

```
    kle->seqh.precision=10;
    kle->seqh.octaves[0]=3;
    kle->seqh.octaves[1]=2;

    kle->frmh.head.description_length=sizeof(KlicsFrameHeader);
    kle->frmh.head.version_number[0]=0;
    kle->frmh.head.version_number[1]=1;

    kle->encd.bpf_in=(2133+temp*160)/8;        /* High = 64000 bits/frame, Poor = 1
    kle->encd.bpf_out=kle->encd.bpf_in;
    kle->encd.buf_size=kle->encd.bpf_in*4;

    kle->encd.quant=16-(space*15)/1023;
    kle->encd.thresh=1.0;
    kle->encd.compare=1.0;
    kle->encd.base[0]=0.10;
    kle->encd.base[1]=0.10;
    kle->encd.base[2]=0.20;
    kle->encd.base[3]=0.50;
    kle->encd.base[4]=1.00;
    kle->encd.intra=still;
    kle->encd.auto_q=true;
    kle->encd.buf_sw=true;
    kle->encd.prevquact=1;
    kle->encd.prevbytes=13;
}

#ifndef DECODER
pascal ComponentResult
KLPreCompress(Handle storage,register CodecCompressParams *p)
{
    ComponentResult      result;
    CodecCapabilities    *capabilities = p->capabilities;
    short                width=(*p->imageDescription)->width+(capabilities->extendW
    short                height=(*p->imageDescription)->height+(capabilities->exten
    Globals              **glob=(Globals **)storage;
    KlicsE               kle=&(*glob)->kle;
    Handle               ext=NewHandle(sizeof(KlicsSeqHeader));

    DebugMsg("\pKLPreCompress");
    HLock(storage);
    if (MemError()!=noErr) return(MemError());
    switch ( (*p->imageDescription)->depth )  {
        case 24:
            capabilities->wantedPixelSize = 32;
            kle->seqh.channels=3;
            if (noErr!=(result=KLGetTab(glob,UseF32)))
                return(result);
            break;
        default:
            return(codecConditionErr);
            break;
    }

    /* Going to use 3 octaves for Y and 2 for UV so the image must be a multiple o

    capabilities->bandMin = height;
    capabilities->bandInc = capabilities->bandMin;

    capabilities->flags=codecCanCopyPrevComp|codecCanCopyPrev;

    (*glob)->scale[0]=0;
    (*glob)->scale[1]=0;
```

```
    (*glob)->scale[2]=0;

    if (noErr!=(result=KLMalloc(glob,height,width,0))) return result;
    KLSetup(p->sequenceID==0,width,height,(*p->imageDescription)->spatialQuality,(

    BlockMove((Ptr)&kle->seqh,*ext,sizeof(KlicsSeqHeader));
    if (noErr!=(result=SetImageDescriptionExtension(p->imageDescription,ext,klicsC
    return result;

    HUnlock(storage);
    DebugMsg("\pKLPreCompress success");
    return(result);
}
#endif

#ifndef ENCODER
pascal long
KLPreDecompress(Handle storage,register CodecDecompressParams *p)
{
    ComponentResult     result;
    CodecCapabilities   *capabilities = p->capabilities;
    Rect                dRect = p->srcRect;
    long                width;
    long                height;
    long                channels;
    Globals             **glob=(Globals **)storage;
    KlicsE              kle;
    Handle              ext;
    OSErr       err;

    DebugMsg("\pKLPreDecompress");
    if ( !TransformRect(p->matrix,&dRect,nil) )
        return(codecConditionErr);

    HLock(storage);
    kle=&(*glob)->kle;
    switch ( (*p->imageDescription)->depth )  {
        case 24:
            switch(p->dstPixMap.pixelSize) {
            case 32:
                capabilities->wantedPixelSize = 32;
                if (p->conditionFlags&codecConditionNewDepth) {
                    if (noErr!=(err=KLGetTab(glob,Use32)))
                        return(err);
                }
                break;
            case 16:
                capabilities->wantedPixelSize = 16;
                if (p->conditionFlags&codecConditionNewDepth) {
                    if (noErr!=(err=KLGetTab(glob,Use16)))
                        return(err);
                }
                break;
            case 8:
                capabilities->wantedPixelSize = 8;
                if (p->conditionFlags&codecConditionNewClut) {
                    if (noErr!=(err=KLGetTab(glob,Use8)))
                        return(err);
                }
                break;
            }
            channels=3;
            break;
```

```
        default:
            return(codecConditionErr);
            break;
    }

    if (noErr!=(result=GetImageDescriptionExtension(p->imageDescription,&ext,klics
    BlockMove(*ext,(Ptr)&kle->seqh,sizeof(KlicsSeqHeader));
    if (channels==1) kle->seqh.channels=1;

    /* Going to use 3 octaves for Y and 2 for UV so the image must be a multiple o

#ifdef HQ
    (*glob)->scale[0]=0; /* Tree scale */
#else
    (*glob)->scale[0]=1; /* Tree scale */
#endif
    width=kle->seqh.sequence_size[0];
    height=kle->seqh.sequence_size[1];

    switch((*glob)->scale[0]) {
    case 1: /* Quarter size internal */
        (*glob)->scale[1]=1;
        if (p->matrix->matrix[0][0]==p->matrix->matrix[1][1])
            switch(p->matrix->matrix[0][0]) {
            case 32768:
                capabilities->flags=codecCanScale;
                capabilities->extendWidth=width/2-dRect.right;
                capabilities->extendHeight=height/2-dRect.bottom;
                (*glob)->scale[2]=1;
                break;
            case 65536:
                capabilities->extendWidth=width-dRect.right;
                capabilities->extendHeight=height-dRect.bottom;
                (*glob)->scale[2]=0;
                break;
            default:
                capabilities->extendWidth=0;
                capabilities->extendHeight=0;
                (*glob)->scale[2]=0;
                break;
            }
        else {
            capabilities->extendWidth=0;
            capabilities->extendHeight=0;
            (*glob)->scale[2]=0;
        }
        break;
    case 0: /* Full size internal */
        if (p->matrix->matrix[0][0]==p->matrix->matrix[1][1])
            switch(p->matrix->matrix[0][0]) {
            case 32768:
                capabilities->flags=codecCanScale;
                capabilities->extendWidth=width/2-dRect.right;
                capabilities->extendHeight=height/2-dRect.bottom;
                (*glob)->scale[1]=1;
                (*glob)->scale[2]=1;
                break;
            case 131072:
                capabilities->flags=codecCanScale;
                capabilities->extendWidth=width*2-dRect.right;
                capabilities->extendHeight=height*2-dRect.bottom;
                (*glob)->scale[1]=0;
                (*glob)->scale[2]=-1;
```

```
                break;
            case 65536:
                capabilities->extendWidth=width-dRect.right;
                capabilities->extendHeight=height-dRect.bottom;
                (*glob)->scale[1]=0;
                (*glob)->scale[2]=0;
                break;
            default:
                capabilities->extendWidth=0;
                capabilities->extendHeight=0;
                (*glob)->scale[1]=0;
                (*glob)->scale[2]=0;
            }
        else {
            capabilities->extendWidth=0;
            capabilities->extendHeight=0;
            (*glob)->scale[1]=0;
            (*glob)->scale[2]=0;
        }
        break;
    }

    capabilities->bandMin = height;
    capabilities->bandInc = capabilities->bandMin;
    capabilities->flags|=codecCanCopyPrev|codecCanCopyPrevComp|codecCanRemapColor;

    if (noErr!=(result=KLMalloc(glob,height,width,capabilities->wantedPixelSize)))

    HUnlock(storage);
    DebugMsg("\pKLPreDecompress success");
    return(result);
}
#endif

/* Test Versions in C - Colour.c */
void    RGB2YUV32(long *pixmap, short *Yc, short *Uc, short *Vc, int area, int wid
void    YUV2RGB32(long *pixmap, short *Yc, short *Uc, short *Vc, int area, int wid
void    YUV2RGB32x2(Ptr table,long *pixmap, short *Yc, short *Uc, short *Vc, int a

/* Assembler versions - Colour.a */
OUT32X2(Ptr table,long *pixmap,short *Y,short *U,short *V,long width,long height,l
OUT32X2D(Ptr table,long *pixmap,short *Y,short *U,short *V,long width,long height,
OUT32(Ptr table,long *pixmap,short *Y,short *U,short *V,long width,long height,lon
OUT32D(Ptr table,long *pixmap,short *Y,short *U,short *V,long width,long height,lo
OUT8X2(Ptr table,long *pixmap,short *Y,short *U,short *V,long width,long height,lo
OUT8(Ptr table,long *pixmap,short *Y,short *U,short *V,long width,long height,long
OUT16X2(Ptr table,long *pixmap,short *Y,short *U,short *V,long width,long height,l
OUT16(Ptr table,long *pixmap,short *Y,short *U,short *V,long width,long height,lon
IN32(Ptr table,long *pixmap,short *Y,short *U,short *V,long width,long height,long

/* Assembler versions - Color2.a */
void    RGB2YUV2(long *pixmap, short *Yc, short *Uc, short *Vc, int area, int widt
void    YUV2RGB2(long *pixmap, short *Yc, short *Uc, short *Vc, int area, int widt
void    YUV2RGB3(long *pixmap, short *Yc, short *Uc, short *Vc, int area, int widt
void    GREY2Y(long *pixmap, short *Yc, int area, int width, int cols);
void    Y2GREY(long *pixmap, short *Yc, int lines, int width, int cols);
void    Y2GGG(long *pixmap, short *Yc, int lines, int width, int cols);

/*YUV2RGB4((*glob)->Table,pixmap,src[0],src[1],src[2],cols*(*desc)->height>>scale,
YUV2RGB5((*glob)->Table,pixmap,src[0],src[1],src[2],cols*(*desc)->height,width>>sc

#pragma parameter __D0 MicroSeconds
```

Engineering:KlicsCode:CompPict:KlicsCodec.c

```
pascal  unsigned long MicroSeconds(void) = {0x4EB0, 0x81E1, 0x64C};

unsigned long   GetTimerValue(unsigned long *TimerRes)
{
    *TimerRes = CLOCKS_PER_SEC;
    return(MicroSeconds());
}

#ifndef DECODER
pascal long
KLBandCompress(Handle storage,register CodecCompressParams *p)
{
#pragma unused(storage)
    Globals             **glob  = (Globals **)storage;
    ImageDescription    **desc = p->imageDescription;
    char                *baseAddr;
    short               rowBytes;
    Rect                sRect;
    long                offsetH,offsetV;
    OSErr               result = noErr;
    short               *src[3],*dst[3];
    long                *pixmap;
    int                 width=(*desc)->width+KLExtendWidth((*desc)->width);
    int                 height=(*desc)->height+KLExtendHeight((*desc)->height);
    int                 hwidth=width>>1,hheight=height>>1;
    int                 bytes;
    KlicsE              kle;
    char                mmuMode=1;
    char                intra[]="\pENC:Intra-mode", inter[]="\pENC:Inter-mode";
    SharedGlobals       *sGlob;

#ifdef PERFORMANCE
    (void)PerfControl(ThePGlobals,true);
#endif

    DebugMsg("\pBandCompress");
    HLock((Handle)glob);
    kle=&(*glob)->kle;
    sGlob=(*glob)->sharedGlob;

    rowBytes = p->srcPixMap.rowBytes & 0x3fff;
    sRect =  p->srcPixMap.bounds;
    switch ( p->srcPixMap.pixelSize ) {
    case 32:
        offsetH = sRect.left<<2;
        break;
    case 16:
        offsetH = sRect.left<<1;
        break;
    case 8:
        offsetH = sRect.left;
        break;
    default:
        result = codecErr;
        DebugMsg("\pError");
        goto bail;
    }
    offsetV = sRect.top * rowBytes;
    baseAddr = p->srcPixMap.baseAddr + offsetH + offsetV;
    pixmap=(long *)baseAddr;

/*  FSMakeFSSpec(0,0,"\pUser:crap001",&fsspec);
    FSpCreate(&fsspec,'????','????',-1);
```

```
FSpOpenDF(&fsspec,fsWrPerm,&fileRefNum);
area=height*rowBytes;
FSWrite(fileRefNum,&area,(long*)pixmap);
FSClose(fileRefNum);*/

src[0]=(*glob)->src[0]; src[1]=(*glob)->src[1]; src[2]=(*glob)->src[2];
dst[0]=(*glob)->dst[0]; dst[1]=(*glob)->dst[1]; dst[2]=(*glob)->dst[2];
switch(kle->seqh.channels) {
case 3:
    IN32(sGlob->tab[UseF32-1],pixmap,src[0],src[1],src[2],width,height,rowByte
    break;
}


/*********************************************
 *
 *  Klics encode
 *
 *********************************************/
#ifdef DEBUG
    if (p->callerFlags&codecFlagUseImageBuffer) DebugStr("\pUseImageBuffer");   /*
    if (p->callerFlags&codecFlagUseScreenBuffer) DebugStr("\pUseScreenBuffer"); /*
    if (p->callerFlags&codecFlagUpdatePrevious) DebugStr("\pUpdatePrevious");   /*
    if (p->callerFlags&codecFlagNoScreenUpdate) DebugStr("\pNoScreenUpdate");    /*
    if (p->callerFlags&codecFlagDontOffscreen) DebugStr("\pDontOffscreen");      /*
    if (p->callerFlags&codecFlagUpdatePreviousComp) DebugStr("\pUpdatePreviousComp
    if (p->callerFlags&codecFlagForceKeyFrame) DebugStr("\pForceKeyFrame");      /*
    if (p->callerFlags&codecFlagOnlyScreenUpdate) DebugStr("\pOnlyScreenUpdate");
#endif

    kle->buf.buf=(unsigned long *)(p->data+sizeof(KlicsFrameHeader));
    kle->encd.intra=(p->temporalQuality==0);
    kle->frmh.frame_number=p->frameNumber;

    bytes=KlicsEncode(src,dst,kle);

    BlockMove((Ptr)&kle->frmh,p->data,sizeof(KlicsFrameHeader));
    bytes+=sizeof(KlicsFrameHeader);

    (*glob)->prev_frame=p->frameNumber;

    p->data+=bytes;
    p->bufferSize=bytes;
    (*p->imageDescription)->dataSize=bytes;

    p->similarity=(kle->encd.intra?0:Long2Fix(244));
    p->callerFlags=0;
/*  p->callerFlags|=codecFlagUsedImageBuffer|(kle->encd.intra?codecFlagUsedNewImag

bail:

    HUnlock((Handle)glob);
#ifdef PERFORMANCE
    if(0!=(result=PerfDump(ThePGlobals,"\pEncode.perf",false,0)))
        return(result);
#endif
    DebugMsg("\pBandCompress success");
    return(result);
}
#endif

/* Display stuff for debugging
    CGrafPtr    wPort, savePort;
```

```
    Rect        rect;
    Str255      str;

    GetPort((GrafPtr *)&savePort);
    GetCWMgrPort(&wPort);
    SetPort((GrafPtr)wPort);
    SetRect(&rect,0,0,50,30);
    ClipRect(&rect);
    EraseRect(&rect);
    NumToString(frmh->frame_number,str);
    MoveTo(0,20);
    DrawString(str);
    if (frmh->flags&KFH_INTRA) {
        SetRect(&rect,0,30,50,65);
        ClipRect(&rect);
        EraseRect(&rect);
        NumToString(frmh->frame_number/24,str);
        MoveTo(0,50);
        DrawString(str);
    }
    SetRect(&rect,-2000,0,2000,2000);
    ClipRect(&rect);
    SetPort((GrafPtr)savePort);*/

#define flag_tree   0x1
#define flag_wave   0x2
#define flag_show   0x4
#define flag_full   0x8
#define DURATION    66666

long    ModeSwitch(Globals *glob,KlicsFrameHeader *frmh)
{
    long    mode=0, i, fps;
    Boolean repeat=glob->prev_frame==frmh->frame_number,
            next=glob->prev_frame+1==frmh->frame_number;
    CGrafPtr    wPort, savePort;
    Rect        rect;
    Str255      str;

    DebugMsg("\pModeSwitch - begin");
    if (frmh->frame_number==0)
        for(i=0;i<15;i++) glob->out[i]=false;
    if (repeat) {
        glob->run_time=0;
        DebugMsg("\pModeSwitch - repeat (end)");
        return(flag_show|flag_full);
    }

    if (next)
        switch(frmh->flags) {
        case KFH_SKIP:
            DebugMsg("\pModeSwitch - next/skip");
            glob->prev_frame=frmh->frame_number;
            if (glob->sys_time>DURATION) {
                glob->run_time=0;
                if (glob->real_frame!=glob->dpy_frame)
                    mode|=flag_wave|flag_show;
            } else {
                unsigned long frame, late;

                frame=glob->run_frame+(glob->sync_time-glob->run_time)/DURATION;
                late=(glob->sync_time-glob->run_time)%DURATION;
                if (frame<=glob->prev_frame && glob->real_frame!=glob->dpy_frame)
```

```
                mode|=flag_wave|flag_show;
/*          if (frame<=glob->prev_frame && late+glob->wave_time+glob->dpy_time
                mode|=flag_wave flag_show:*/
            }
            break;
        case KFH_INTRA:
            DebugMsg("\pModeSwitch - next/intra");
            mode=flag_tree;
            glob->prev_frame=frmh->frame_number;
            glob->real_frame=glob->prev_frame;
            if (glob->sys_time>DURATION) {
                glob->run_time=0;
                mode|=flag_wave|flag_show|flag_full;
            } else
/*              if (glob->run_time==0) {*/
                    glob->key_time=glob->sync_time-glob->run_time;
                    glob->run_time=glob->sync_time-glob->sys_time;
                    glob->run_frame=glob->prev_frame;
                    mode|=flag_wave|flag_show|flag_full;
/*              } else {
                    unsigned long frame, late;

                    frame=glob->run_frame+(glob->sync_time-glob->run_time)/DURATIO
                    late=(glob->sync_time-glob->run_time)%DURATION;
                    if (frame<=glob->prev_frame)
                        mode|=flag_wave|flag_show|flag_full:
                }*/
            break;
        default:
            DebugMsg("\pModeSwitch - next/inter");
            mode=flag_tree;
            glob->prev_frame=frmh->frame_number;
            glob->real_frame=glob->prev_frame;
            if (glob->sys_time>DURATION) {
                glob->run_time=0;
                mode|=flag_wave|flag_show;
            } else
                if (glob->run_time==0) {
                    glob->run_time=glob->sync_time-glob->sys_time;
                    glob->run_frame=glob->prev_frame;
                    mode|=flag_wave|flag_show;
                } else {
                    unsigned long frame, late;

                    frame=glob->run_frame+(glob->sync_time-glob->run_time)/DURATIO
                    late=(glob->sync_time-glob->run_time)%DURATION;
                    if (frame<=glob->prev_frame)
                        mode|=flag_wave|flag_show;
/*                  if (frame<=glob->prev_frame && late+glob->tree_time+glob->wave
                        mode|=flag_wave|flag_show;*/
                }
            break;
        }
    else
        switch(frmh->flags) {
        case KFH_SKIP:
            DebugMsg("\pModeSwitch - jump/skip");
            glob->run_time=0;
            break;
        case KFH_INTRA:
            DebugMsg("\pModeSwitch - jump/intra");
            mode=flag_tree|flag_wave|flag_show|flag_full;
            for(i=glob->prev_frame;i<frmh->frame_number;i++)
```

```
            glob->out[frmh->frame_number%15]=0;
            glob->prev_frame=frmh->frame_number;
            glob->real_frame=glob->prev_frame;
            glob->run_time=0;
            break;
        default:
            DebugMsg("\pModeSwitch - jump/inter");
            glob->run_time=0;
            break;
        }
    DebugMsg("\pModeSwitch - display info");
#ifndef COMPONENT
/*  glob->out[frmh->frame_number%15]=(mode&flag_show)!=0;
    for(i=0,fps=0;i<15;i++) if (glob->out[i]) fps++;
    GetPort((GrafPtr *)&savePort);
    GetCWMgrPort(&wPort);
    SetPort((GrafPtr)wPort);
    SetRect(&rect,0,20,120,50);
    ClipRect(&rect);
    EraseRect(&rect);
    NumToString(frmh->frame_number,str);
    MoveTo(0,35);
    DrawString(str);
    DrawString("\p:");
    NumToString(fps,str);
    DrawString(str);
    MoveTo(0,50);
    for(i=0;i<15;i++)
        if (glob->out[i]) DrawString("\pX");
        else DrawString("\pO");
    SetRect(&rect,-2000,0,2000,2000);
    ClipRect(&rect);
    SetPort((GrafPtr)savePort);*/
#endif
    DebugMsg("\pModeSwitch - end");
    return(mode);
}

#ifndef ENCODER
pascal long
KLBandDecompress(Handle storage,register CodecDecompressParams *p)
{
#pragma unused(storage)
    Globals **glob = (Globals **)storage;
    ImageDescription    **desc = p->imageDescription;
    int                 x,y;
    char                *baseAddr;
    short               rowBytes;
    Rect                dRect;
    long                offsetH,offsetV;
    OSErr               result = noErr;
    short               *src[3],*dst[3];
    long                *pixmap;
    int                 width=(*desc)->width+KLExtendWidth((*desc)->width);
    int                 height=(*desc)->height+KLExtendHeight((*desc)->height);
    int                 hwidth=width>>1,hheight=height>>1,area=height*width;
    KlicsE              kle;
    KlicsFrameHeader    *frmh;
    char                mmuMode=1;
    long                mode;
    SharedGlobals       *sGlob;
/*
    FILE                *fp;
```

```
    char                file_name[30];
    CGrafPtr            wPort, savePort;
    Rect                rect;
    Str255              str;
*/

    HLock((Handle)glob);
    DebugMsg("\pBandDecompress");
    (*glob)->sys_time=GetTimerValue(&(*glob)->sys_time);
    (*glob)->sys_time-=(*glob)->sync_time;

#ifdef PERFORMANCE
    (void)PerfControl(ThePGlobals,true);
#endif

    kle=&(*glob)->kle;
    sGlob=(*glob)->sharedGlob;

    dRect = p->srcRect;
    if ( !TransformRect(p->matrix,&dRect,nil) ) {
        DebugMsg("\pTransformRect Error");
        return(paramErr);
    }
    rowBytes = p->dstPixMap.rowBytes & 0x3fff;
    offsetH = (dRect.left - p->dstPixMap.bounds.left);
    switch (  p->dstPixMap.pixelSize ) {
    case 32:
        offsetH <<=2;
        break;
    case 16:
        offsetH <<=1;
        break;
    case 8:
        break;
    default:
        result = codecErr;
        DebugMsg("\pDepth Error");
        goto bail;
    }
    offsetV = (dRect.top - p->dstPixMap.bounds.top) * rowBytes;
    baseAddr = p->dstPixMap.baseAddr + offsetH + offsetV;
    pixmap=(long *)baseAddr;

    /*********************************************
     *
     *  Klics decode
     *
     *********************************************/

    src[0]=(*glob)->src[0]; src[1]=(*glob)->src[1]; src[2]=(*glob)->src[2];
    dst[0]=(*glob)->dst[0]; dst[1]=(*glob)->dst[1]; dst[2]=(*glob)->dst[2];

    frmh=(KlicsFrameHeader *)p->data;
    kle->buf.buf=(unsigned long *)(p->data+sizeof(KlicsFrameHeader));
    mode=ModeSwitch(*glob,frmh);

    KlicsDecode(src,dst,&kle->seqh,frmh,&kle->buf,mode,(*glob)->scale,&(*glob)->tr

    if ( kle->buf.ptr-kle->buf.buf > frmh->length+2)
        DebugMsg("\pWarning: Decompressor read passed end of buffer");

    p->data[0]='X';
    p->data[1]=mode&flag_tree?'T':' ';
```

```
    p->data[2]=mode&flag_wave?'W':' ';
    p->data[3]=mode&flag_show?'S':' ';
    p->data[4]=mode&flag_full?'F':' ';
    p->data[5]=frmh->flags&KFH_INTRA?'I':' ';
    p->data[6]=frmh->flags&KFH_SKIP?'K':' ';
    p->data[7]='X';

    p->data-=p->bufferSize;

    /*********************************************
     *
     *  signed 10 bit YUV-unsigned 8 RGB convert
     *
     *********************************************/

#ifdef COMPONENT
    SwapMMUMode(&mmuMode);
#endif
    if (mode&flag_show) {
        (*glob)->sync_time=GetTimerValue(&(*glob)->sync_time);
        (*glob)->dpy_frame=(*glob)->real_frame;
        if ((*glob)->scale[2]<(*glob)->scale[1]) {
            switch(kle->seqh.channels) {
            case 3:
                switch (p->dstPixMap.pixelSize) {
                case 32:
                    if (mode&flag_full)
                        OUT32X2(sGlob->tab[Use32-1],pixmap,src[0],src[1],src[2],wi
                    else
                        OUT32X2D(sGlob->tab[Use32-1],pixmap,src[0],src[1],src[2],w
                    break;
                case 16:
                    OUT16X2(sGlob->tab[Use16-1],pixmap,src[0],src[1],src[2],width>
                    break;
                case 8:
                    OUT8X2(sGlob->tab[Use8-1],pixmap,src[0],src[1],src[2],width>>(
                    break;
                }
                break;
            }
        } else {
            switch(kle->seqh.channels) {
            case 3:
                switch (p->dstPixMap.pixelSize) {
                case 32:
                    if (mode&flag_full)
                        OUT32(sGlob->tab[Use32-1],pixmap,src[0],src[1],src[2],widt
                    else
                        OUT32D(sGlob->tab[Use32-1],pixmap,src[0],src[1],src[2],wid
                    break;
                case 16:
                    OUT16(sGlob->tab[Use16-1],pixmap,src[0],src[1],src[2],width>>(
                    break;
                case 8:
                    OUT8(sGlob->tab[Use8-1],pixmap,src[0],src[1],src[2],width>>(*g
                    break;
                }
                break;
            }
        }
        (*glob)->dpy_time=GetTimerValue(&(*glob)->dpy_time);
        (*glob)->dpy_time-=(*glob)->sync_time;
    }
```

```
    CLEARA2();
    (*glob)->sync_time=GetTimerValue(&(*glob)->sync_time);

#ifdef COMPONENT
    SwapMMUMode(&mmuMode);
#endif

bail:
    HUnlock((Handle)glob);

#ifdef PERFORMANCE
    if(0!=(result=PerfDump(ThePGlobals,"\pDecode.perf",false,0)))
        return(result);
#endif
    DebugMsg("\pBandDecompress success");
    return(result);
}
#endif
```

```
/*******************************
 *
 *  © Copyright 1993 KLICS Limited
 *  All rights reserved.
 *
 *  Written by: Adrian Lewis
 *
 *******************************/
/*
 *  Second generation header file
 */

#include    <stdio.h>

/* useful X definitions */
/*typedef   char    Boolean;*/
typedef char    *String;
#define True    1
#define False   0

/* new Blk definition */
typedef int     Blk[4];

#define WT_Haar  0
#define WT_Daub4 1

/* mode constructors */
#define M_LPF   1
#define M_STILL 2
#define M_SEND  4
#define M_STOP  8
#define M_VOID  16
#define M_QUIT  32

/* LookAhead histogram */
#define HISTO   300
#define HISTO_DELTA 15.0
#define HISTO_BITS 10

/* Fast Functions */

/* Is the block all zero ? */
#define BlkZero(block) \
    block[0]==0 && block[1]==0 && block[2]==0 && block[3]==0

/* Sum of the absolute values */
#define Decide(new) \
    abs(new[0])+ \
    abs(new[1])+ \
    abs(new[2])+ \
    abs(new[3])

/* Sum of the absolute differences */
#define DecideDelta(new,old) \
    abs(new[0]-old[0])+ \
    abs(new[1]-old[1])+ \
    abs(new[2]-old[2])+ \
    abs(new[3]-old[3])

/* Adjust the norm for comparison with SigmaAbs */
#define DecideDouble(norm) (4.0*norm)

/* Get addresses from x,y coords of block, sub-band, octave,
```

```
 * image size and mask (directly related to octave) information
 */
#define GetAddr(addr,x,y,sub,oct,size,mask) \
{ int    smask=mask>>1, \
         x0=x|(sub&1?smask:0), \
         x1=x|(sub&1?smask:0)|mask, \
         y0=(y|(sub&2?smask:0))*size[0], \
         y1=(y|(sub&2?smask:0)|mask)*size[0]; \
 \
    addr[0]=x0+y0; \
    addr[1]=x1+y0; \
    addr[2]=x0+y1; \
    addr[3]=x1+y1; \
}

/* Get data values from addresses and memory */
#define GetData(addr,block,data) \
    block[0]=(int)data[addr[0]]; \
    block[1]=(int)data[addr[1]]; \
    block[2]=(int)data[addr[2]]; \
    block[3]=(int)data[addr[3]];

#define VerifyData(block,mask,tmp) \
    tmp=block&mask; \
    if (tmp!=0 && tmp!=mask) { \
        block=block<0?mask:-mask; \
    }

/* Put data values to memory using addresses */
#define PutData(addr,block,data) \
    data[addr[0]]=(short)block[0]; \
    data[addr[1]]=(short)block[1]; \
    data[addr[2]]=(short)block[2]; \
    data[addr[3]]=(short)block[3];

/* Put zero's to memory using addresses */
#define PutZero(addr,data) \
    data[addr[0]]=0; \
    data[addr[1]]=0; \
    data[addr[2]]=0; \
    data[addr[3]]=0;

/* Mode: M_VOID Put zero's and find new mode */
#define DoZero(addr,dst,mode,oct) \
    PutZero(addr,dst); \
    mode[oct]=oct==0?M_STOP:M_VOID;

/* Descend the tree structure
 * Copy mode, decrement octave (& mask), set branch to zero
 */
#define DownCounters(mode,oct,mask,blk) \
    mode[oct-1]=mode[oct]; \
    oct--; \
    mask = mask>>1; \
    blk[oct]=0;

/* Ascend the tree structure
 * Ascend tree (if possible) until branch not 3
 * If at top then set mode to M_QUIT
 * Else increment branch and x, y coords
 */
#define StopCounters(mode,oct,mask,blk,x,y,octs) \
    while(oct<octs-1 && blk[oct]==3) { \
```

Engineering:KlicsCode:CompPict:Klics.h

```
        blk[oct]=0; \
        mask= mask<<1; \
        x &= -mask; \
        y &= -mask; \
        oct++; \
    } \
    if (oct==octs-1) mode[oct]=M_QUIT; \
    else { \
        blk[oct]++; \
        x ^= mask<<1; \
        if (blk[oct]==2) y ^= mask<<1; \
        mode[oct]=mode[oct+1]; \
    }
```

Engineering:KlicsCode:CompPict:Haar.a

```
*------------------------------------
*
*   © Copyright 1993 KLICS Limited
*   All rights reserved.
*
*   Written by: Adrian Lewis
*
*------------------------------------
*
*   68000 FastForward/Backward Haar
*
*------------------------------------
        macro
        Fwd0        &addr0,&dG,&dH
*
        move.w      (&addr0),&dG        ; dG=*(short *)addr1
        move.w      &dG,&dH             ; dH=dG
*
        endm
*------------------------------------
        macro
        Fwd1        &addr1,&addr0,&dG,&dH
*
        move.w      (&addr1),d0         ; v=*(short *)addr2
        add.w       d0,&dH              ; dH+=v
        sub.w       d0,&dG              ; dG-=v
        clr.w       d0                  ; d0=0
        asr.w       #1,&dH              ; dAH>>=1
        addx.w      d0,&dH              ; round dH
        asr.w       #1,&dG              ; dG>>=1
        addx.w      d0,&dG              ; round dG
        move.w      &dH,(&addr0)        ; *(short *)addr0=dH
        move.w      &dG,(&addr1)        ; *(short *)addr1=dG
*
        mend
*------------------------------------
        macro
        Fwd         &base,&end,&inc
*
        movea.l     &base,a0                    ; addr0=base
        move.l      &inc,d0                     ; d0=inc
        asr.l       #1,d0                       ; d0=inc>>1
        movea.l     a0,a1                       ; addr1=addr0
        suba.l      d0,a1                       ; addr1-=(inc>>1)
@do     Fwd0        a0,d4,d5                    ; Fwd0(addr0,dG,dH)
        adda.l      &inc,a1                     ; addr1+=inc
        Fwd1        a1,a0,d4,d5                 ; Fwd1(addr1,addr0,dG,dH)
        adda.l      &inc,a0                     ; addr0+=inc
        cmpa.l      a0,&end                     ; addr0<end
        bgt.s       @do                         ; while
*
        endm
*------------------------------------
HaarForward FUNC    EXPORT
*
        link        a6,#0                       ; no local variables
        movem.l     d4-d7/a3-a5,-(a7)           ; store registers
*
        move.l      $000C(a6),d3                ; inc=inc1
        movea.l     $0008(a6),a5                ; base=data
        move.l      $0010(a6),d6                ; end1
        move.l      $0018(a6),d7                ; end2
        move.l      $0014(a6),d2                ; inc2
```

Engineering:KlicsCode:CompPict:Haar.a

```
@do     movea.l   a5,a4                 ; end=base
        adda.l    d6,a4                 ; end+=endl
        Fwd       a5,a4,d3              ; Fwd(base,end,inc)
        adda.l    d2,a5                 ; base+=inc2
        cmpa.l    d7,a5                 ; end2>base
        blt.s     @do                   ; for
*
        movem.l   (a7)+,d4-d7/a3-a5     ; restore registers
        unlk      a6                    ; remove locals
        rts                             ; return
*
        ENDFUNC
*-----------------------------------
        macro
        Bwd0      &addr0,&dG,&dH
*
        move.w    (&addr0),&dG    ; dG=*(short *)addr0
        move.w    &dG,&dH         ; dH=dG
*
        endm
*-----------------------------------
        macro
        Bwd1      &addr1,&addr0,&dG,&dH
*
        move.w    (&addr1),d0     ; v=*(short *)addr1
        add.w     d0,&dH          ; dH+=v
        sub.w     d0,&dG          ; dG-=v
        move.w    &dH,(&addr0)    ; *(short *)addr0=dH
        move.w    &dG,(&addr1)    ; *(short *)addr1=dG
*
        endm
*-----------------------------------
        macro
        Bwd       &base,&count,&inc
*
        movea.l   &base,a0              ; addr0=base
        move.l    &inc,d0               ; d0=inc
        asr.l     #1,d0                 ; d0=inc>>1
        movea.l   a0,a1                 ; addr1=addr0
        suba.l    d0,a1                 ; addr1-=(inc>>1)
@do     Bwd0      a0,d4,d5              ; Bwd0(addr0,dG,dH)
        adda.l    &inc,a1               ; addr1+=inc
        Bwd1      a1,a0,d4,d5           ; Bwd1(addr1,addr0,dG,dH)
        adda.l    &inc,a0               ; addr0+=inc
        dbf       &count,@do            ; while -1!=count
*
        endm
*-----------------------------------
HaarBackward    FUNC    EXPORT
*
*  d0 - spare, d1 - count1, d2 - inc2, d3 - inc1, d4 - dG, d5 - dH, d6 - loop1, d
*
        link      a6,#0                 ; no local variables
        movem.l   d4-d7/a3-a5,-(a7)     ; store registers
*
        move.l    $000C(a6),d3          ; inc=inc1
        movea.l   $0008(a6),a5          ; base=data
        move.l    $0010(a6),d6          ; loop1 (width/height)
        move.l    $0018(a6),d7          ; loop2 (height/width)
        move.l    $0014(a6),d2          ; inc2
        subq.l    #1,d7                 ; loop2-=1
        lsr.l     #1,d6                 ; loop1/=2
        subq.l    #1,d6                 ; loop1-=1
```

Engineering:KlicsCode:CompPict:Haar.a

```
@do     move.l      d6,d1               ; count1=loop1
        Bwd         a5,d1,d3            ; Bwd(base,count,inc)
        adda.l      d2,a5               ; base+=inc2
        dbf         d7,@do              ; while -1!=--loop2
*
        movem.l     (a7)+,d4-d7/a3-a5   ; restore registers
        unlk        a6                  ; remove locals
        rts                             ; return
*
        ENDFUNC
*------------------------------------
HaarXTopBwd FUNC    EXPORT
*
        link        a6,#0               ; no local variables
*
        movea.l     $0008(a6),a0        ; start
        move.l      $000C(a6),d3        ; area
        lsr.l       #1,d3               ; area (long)
        subq.l      #1,d3               ; area-=1
@do     move.l      (a0),d0             ; d0=HG=*Y
        move.l      d0,d1               ; d1=HG
        swap        d1                  ; d1=GH
        neg.w       d0                  ; d0=H(-G)
        add.l       d1,d0               ; d0=01
        move.l      d0,(a0)+            ; *Y++=01
        dbf         d3,@do              ; while -1!=--area
*
        unlk        a6                  ; remove locals
        rts                             ; return
*
        ENDFUNC
*------------------------------------
HaarTopBwd  FUNC    EXPORT
*
        link        a6,#0               ; no local variables
        movem.l     d4-d6,-(a7)         ; store registers
*
        movea.l     $0008(a6),a0        ; startH
        movea.l     a0,a1               ; startG
        move.l      $000C(a6),d4        ; height
        move.l      $0010(a6),d3        ; width
        move.l      d3,d6               ; linelen=width
        add.l       d6,d6               ; linelen (bytes)
        lsr.l       #1,d4               ; height/=2
        lsr.l       #1,d3               ; width/=2
        subq.l      #1,d4               ; height-=1
        subq.l      #1,d3               ; width-=1
@do1    adda.l      d6,a1               ; startG+=linelen
        move.l      d3,d5               ; linecount=width
@do2    move.l      (a0),d0             ; d0=HAHB=*Y0
        move.l      (a1),d1             ; d1=GAGB=*Y1
        move.l      d0,d2               ; d2=HAHB
        add.l       d1,d0               ; d0=0A0B
        sub.l       d1,d2               ; d2=1A1B
*
        move.l      d0,d1               ; d1=HG
        swap        d1                  ; d1=GH
        neg.w       d0                  ; d0=H(-G)
        add.l       d1,d0               ; d0=01
        move.l      d0,(a0)+            ; *Y0++=0A0B
*
        move.l      d2,d1               ; d1=HG
        swap        d1                  ; d1=GH
```

Engineering:KlicsCode:CompPict:Haar.a

```
        neg.w     d2                 ; d2=H(-G)
        add.l     d1,d2              ; d2=01
        move.l    d2,(a1)+           ; *Y1++=1A1B

        dbf       d5,@do2            ; while -1!=--linecount
        move.l    a1,a0              ; startH=startG
        dbf       d4,@do1            ; while -1!=--height

        movem.l   (a7)+,d4-d6        ; restore registers
        unlk      a6                 ; remove locals
        rts                          ; return

        ENDFUNC
*---------------------------------
        END
```

Engineering:KlicsCode:CompPict:ConvolveSH3.c

```
/******************************
 *
 * © Copyright 1993 KLICS Limited
 * All rights reserved.
 *
 * Written by: Adrian Lewis
 *
 ******************************/
/*
    2D wavelet transform convolver (fast hardware emulation)
    New improved wavelet coeffs : 11 19 5 3

    Optimized for speed:
        dirn - False
        src/dst octave == 0
*/

#define FwdS(addr0,dAG,dAH) \
    v=*(short *)addr0; \
    dAG=(v3=v+(vs=v<<1)); \
    dAG+=v+(vs<<=1); \
    dAH=v3+(vs<<=1); \
    dAH+=v3+(vs<<=1);

#define Fwd1(addr1,dAG,dAH,dBG,dBH) \
    v=*(short *)addr1; \
    dBG=(v3=v+(vs=v<<1)); \
    dAH+=v+(vs<<=1); \
    dBH=v3+(vs<<=1); \
    dAG-=v3+(vs<<=1);

#define Fwd2(addr2,addr1,addr0,dAG,dAH,dBG,dBH) \
    v=*(short *)addr2; \
    dAH-=(v3=v+(vs=v<<1)); \
    dBG+=v+(vs<<=1); \
    dAG+=v3+(vs<<=1); \
    dBH+=v3+(vs<<=1); \
    *(short *)addr0=(dAH+15)>>5; \
    *(short *)addr1=(dAG+15)>>5;

#define Fwd3(addr3,dAG,dAH,dBG,dBH) \
    v=*(short *)addr3; \
    dAG=(v3=v+(vs=v<<1)); \
    dBH+=v+(vs<<=1); \
    dAH=v3+(vs<<=1); \
    dBG-=v3+(vs<<=1);

#define Fwd0(addr0,addr3,addr2,dAG,dAH,dBG,dBH) \
    v=*(short *)addr0; \
    dBH-=(v3=v+(vs=v<<1)); \
    dAG+=v+(vs<<=1); \
    dBG+=v3+(vs<<=1); \
    dAH+=v3+(vs<<=1); \
    *(short *)addr2=(dBH+15)>>5; \
    *(short *)addr3=(dBG+15)>>5;

#define FwdE(addr3,addr2,dBG,dBH) \
    v=*(short *)addr3; \
    dBH+=(vs=v<<1); \
    dBG-=(vs<<2); \
    *(short *)addr2=(dBH+15)>>5; \
    *(short *)addr3=(dBG+15)>>5;
```

Engineering:KlicsCode:CompPict:ConvolveSH3.c

```
#define Fwd(base,end,inc) \
    addr0=base; \
    addr3=addr0-(inc>>2); \
    addr2=addr3-(inc>>2); \
    addr1=addr2-(inc>>2); \
    FwdS(addr0,dAG,dAH); \
    addr1+=inc; \
    Fwd1(addr1,dAG,dAH,dBG,dBH); \
    addr2+=inc; \
    Fwd2(addr2,addr1,addr0,dAG,dAH,dBG,dBH); \
    addr3+=inc; \
    while(addr3<end) { \
        Fwd3(addr3,dAG,dAH,dBG,dBH); \
        addr0+=inc; \
        Fwd0(addr0,addr3,addr2,dAG,dAH,dBG,dBH); \
        addr1+=inc; \
        Fwd1(addr1,dAG,dAH,dBG,dBH); \
        addr2+=inc; \
        Fwd2(addr2,addr1,addr0,dAG,dAH,dBG,dBH); \
        addr3+=inc; \
    } \
    FwdE(addr3,addr2,dBG,dBH);

extern  void FASTFORWARD(char *data, long inc1, long end1, long inc2, char *end2);
extern  void HAARFORWARD(char *data, long inc1, long end1, long inc2, char *end2);

void    FastForward(char *data, long inc1, long end1, long inc2, char *end2)
{
    register short  v, vs, v3, dAG, dAH, dBG, dBH, inc;
    register char   *addr0, *addr1, *addr2, *addr3, *end;
    char    *base;

    inc=inc1;
    for(base=data;base<end2;base+=inc2) {
        end=base+end1;
        Fwd(base,end,inc);
    }
}


void    Daub4Forward(short *data, int size[2], int oct_dst)
{
    int     oct, area=size[0]*size[1]<<1;
    short   width=size[0]<<1;
    char    *top=area+(char *)data, *left=width+(char *)data;

    for(oct=0;oct!=oct_dst;oct++) {
        long    cinc=2<<oct, cinc4=cinc<<2,
                rinc=size[0]<<oct+1, rinc4=rinc<<2; /* col and row increments in t

        FASTFORWARD((char *)data,cinc4,width-cinc,rinc,top);
        FASTFORWARD((char *)data,rinc4,area-rinc,cinc,left);
    }
}

void    HaarForward(short *data, int size[2], int oct_dst)
{
    int     oct, area=size[0]*size[1]<<1;
    short   width=size[0]<<1;
    char    *top=area+(char *)data, *left=width+(char *)data;

    for(oct=0;oct!=oct_dst;oct++) {
        long    cinc=2<<oct, cinc2=cinc<<1,
```

Engineering:KlicsCode:CompPict:ConvolveSH3.c

```
            rinc=size[0]<<oct+1, rinc2=rinc<<1; /* col and row increments in t

        HAARFORWARD((char *)data,cinc2,width,rinc,top);
        HAARFORWARD((char *)data,rinc2,area,cinc,left);
    }

void    HybridForward(short *data, int size[2], int oct_dst)
{
    int     oct, area=size[0]*size[1]<<1;
    short   width=size[0]<<1;
    char    *top=area+(char *)data, *left=width+(char *)data;

    HAARFORWARD((char *)data,4,width,size[0]<<1,top);
    HAARFORWARD((char *)data,size[0]<<2,area,2,left);
    for(oct=1;oct!=oct_dst;oct++) {
        long    cinc=2<<oct, cinc4=cinc<<2,
                rinc=size[0]<<oct+1, rinc4=rinc<<2; /* col and row increments in t

        FASTFORWARD((char *)data,cinc4,width-cinc,rinc,top);
        FASTFORWARD((char *)data,rinc4,area-rinc,cinc,left);
    }
}

#define BwdS0(addr0,dAG,dAH,dBH) \
    v=*(short *)addr0; \
    dAG= -(v3=v+(vs=v<<1)); \
    dAH=v+(vs<<=1); \
    dBH=vs<<1; \

#define BwdS1(addr1,addr0,dAG,dAH,dBH) \
    v=*(short *)addr1; \
    dBH+=(vs=v<<1); \
    v3=vs+v; \
    dAG+=v3+(vs<<=2); \
    dAH-=v3+(vs<<=1); \
    *(short *)addr0=(dBH+3)>>3;

#define Bwd2(addr2,dAG,dAH,dBG,dBH) \
    v=*(short *)addr2; \
    dBG= -(v3=v+(vs=v<<1)); \
    dBH=v+(vs<<=1); \
    dAH+=v3+(vs<<=1); \
    dAG+=v3+(vs<<=1);

#define Bwd3(addr3,addr2,addr1,dAG,dAH,dBG,dBH) \
    v=*(short *)addr3; \
    dAH+=(v3=v+(vs=v<<1)); \
    dAG+=v+(vs<<=1); \
    dBG+=v3+(vs<<=1); \
    dBH-=v3+(vs<<=1); \
    *(short *)addr1=(dAH+7)>>4; \
    *(short *)addr2=(dAG+7)>>4;

#define Bwd0(addr0,dAG,dAH,dBG,dBH) \
    v=*(short *)addr0; \
    dAG= -(v3=v+(vs=v<<1)); \
    dAH=v+(vs<<=1); \
    dBH+=v3+(vs<<=1); \
    dBG+=v3+(vs<<=1);

#define Bwd1(addr1,addr0,addr3,dAG,dAH,dBG,dBH) \
    v=*(short *)addr1; \
```

```
    dBH+=(v3=v+(vs=v<<1)); \
    dBG+=v+(vs<<=1); \
    dAG+=v3+(vs<<=1); \
    dAH-=v3+(vs<<=1); \
    *(short *)addr3=(dBH+7)>>4; \
    *(short *)addr0=(dBG+7)>>4;

#define BwdE2(addr2,dAG,dAH,dBH) \
    v=*(short *)addr2; \
    v3=v+(vs=v<<1); \
    dBH=(vs<<=2); \
    dAH+=v3+vs; \
    dAG+=v3+(vs<<=1);

#define BwdE3(addr3,addr2,addr1,dAG,dAH,dBH) \
    v=*(short *)addr3; \
    dAH+=(v3=v+(vs=v<<1)); \
    dAG+=v+(vs<<=1); \
    dBH-=v3+(vs<<=1); \
    dBH-=v3+(vs<<=1); \
    *(short *)addr1=(dAH+7)>>4; \
    *(short *)addr2=(dAG+7)>>4; \
    *(short *)addr3=(dBH+3)>>3;

#define Bwd(base,end,inc) \
    addr0=base; \
    addr3=addr0-(inc>>2); \
    addr2=addr3-(inc>>2); \
    addr1=addr2-(inc>>2); \
    BwdS0(addr0,dAG,dAH,dBH); \
    addr1+=inc; \
    BwdS1(addr1,addr0,dAG,dAH,dBH); \
    addr2+=inc; \
    while(addr2<end) { \
        Bwd2(addr2,dAG,dAH,dBG,dBH); \
        addr3+=inc; \
        Bwd3(addr3,addr2,addr1,dAG,dAH,dBG,dBH); \
        addr0+=inc; \
        Bwd0(addr0,dAG,dAH,dBG,dBH); \
        addr1+=inc; \
        Bwd1(addr1,addr0,addr3,dAG,dAH,dBG,dBH); \
        addr2+=inc; \
    } \
    BwdE2(addr2,dAG,dAH,dBH); \
    addr3+=inc; \
    BwdE3(addr3,addr2,addr1,dAG,dAH,dBH);

extern void FASTBACKWARD(char *data, long incl, long loop1, long inc2, char *end2)
extern void HAARBACKWARD(char *data, long incl, long loop1, long inc2, long loop2)
extern void HAARTOPBWD(char *data,long height,long width);
/* extern void  HAARXTOPBWD(char *data,long area);*/

void    FastBackward(char *data, long incl, long end1, long inc2, char *end2)
{
    register short  v, vs, v3, dAG, dAH, dBG, dBH, inc;
    register char   *addr0, *addr1, *addr2, *addr3, *end;
    char    *base;

    inc=incl;
    for(base=data;base<end2;base+=inc2) {
        end=base+end1;
        Bwd(base,end,inc);
    }
```

```
}

void    Daub4Backward(short *data,int size[2],int oct_src)
{
    int     oct, area=size[0]*size[1]<<1;
    short   width=size[0]<<1;
    char    *top=area+(char *)data, *left=width+(char *)data;

    for(oct=oct_src-1;oct>=0;oct--) {
        long    cinc=2<<oct, cinc4=cinc<<2,
                rinc=size[0]<<oct+1, rinc4=rinc<<2; /* col and row increments in t

        FASTBACKWARD((char *)data,rinc4,area-(rinc<<1),cinc,left);
        FASTBACKWARD((char *)data,cinc4,width-(cinc<<1),rinc,top);
    }
}

void    HaarBackward(data,size,oct_src)

short   *data;
int     size[2], oct_src;

{
    int     oct, area=size[0]*size[1]<<1;
    short   width=size[0]<<1;
    char    *top=area+(char *)data, *left=width+(char *)data;

    for(oct=oct_src-1;oct>0;oct--) {
        long    cinc=2<<oct, cinc2=cinc<<1,
                rinc=size[0]<<oct+1, rinc2=rinc<<1; /* col and row increments in t

        HAARBACKWARD((char *)data,rinc2,size[1]>>oct,cinc,size[0]>>oct);
        HAARBACKWARD((char *)data,cinc2,size[0]>>oct,rinc,size[1]>>oct);
    }
    HAARTOPBWD((char *)data,size[1],size[0]);
/*  HAARXTOPBWD((char *)data,area>>1);*/
}

void    HybridBackward(data,size,oct_src)

short   *data;
int     size[2], oct_src;

{
    int     oct, area=size[0]*size[1]<<1;
    short   width=size[0]<<1;
    char    *top=area+(char *)data, *left=width+(char *)data;

    for(oct=oct_src-1;oct>0;oct--) {
        long    cinc=2<<oct, cinc4=cinc<<2,
                rinc=size[0]<<oct+1, rinc4=rinc<<2; /* col and row increments in t

        FASTBACKWARD((char *)data,rinc4,area-(rinc<<1),cinc,left);
        FASTBACKWARD((char *)data,cinc4,width-(cinc<<1),rinc,top);
    }
    HAARTOPBWD((char *)data,size[1],size[0]);
/*  HAARXTOPBWD((char *)data,area>>1);*/
}
```

Engineering:KlicsCode:CompPict:ConvolveSH3.a

```
*-------------------------------------
*
*   © Copyright 1993 KLICS Limited
*   All rights reserved.
*
*   Written by: Adrian Lewis
*
*-------------------------------------
*
*   68000 FastForward/Backward code
*
*-------------------------------------
        seg         'klics'

        macro
        FwdStart    &addr0,&dAG,&dAH
*
        move.w      (&addr0),d0     ; v=*(short *)addr0
        move.w      d0,d1           ; vs=v
        add.w       d1,d1           ; vs<<=1
        move.w      d1,d2           ; v3=vs
        add.w       d0,d2           ; v3=vs+v
        move.w      d2,&dAG         ; dAG=v3
        add.w       d1,d1           ; vs<<=1
        add.w       d0,&dAG         ; dAG+=v
        add.w       d1,&dAG         ; dAG+=vs
        move.w      d2,&dAH         ; dAH=v3
        add.w       d1,d1           ; vs<<=1
        add.w       d1,&dAH         ; dAH+=vs
        add.w       d2,&dAH         ; dAH+=v3
        add.w       d1,d1           ; vs<<=1
        add.w       d1,&dAH         ; dAH+=vs
*
        endm
*-------------------------------------
        macro
        FwdOdd      &addr1,&dAG,&dAH,&dBG,&dBH
*
        move.w      (&addr1),d0     ; v=*(short *)addr1
        move.w      d0,d1           ; vs=v
        add.w       d1,d1           ; vs<<=1
        move.w      d1,d2           ; v3=vs
        add.w       d0,d2           ; v3=vs+v
        move.w      d2,&dBG         ; dBG=v3
        add.w       d1,d1           ; vs<<=1
        add.w       d0,&dAH         ; dAH+=v
        add.w       d1,&dAH         ; dAH+=vs
        move.w      d2,&dBH         ; dBH=v3
        add.w       d1,d1           ; vs<<=1
        add.w       d1,&dBH         ; dBH+=vs
        sub.w       d2,&dAG         ; dAG-=v3
        add.w       d1,d1           ; vs<<=1
        sub.w       d1,&dAG         ; dAG-=vs
*
        endm
*-------------------------------------
        macro
        FwdEven     &addr2,&addr1,&addr0,&dAG,&dAH,&dBG,&dBH
*
        move.w      (&addr2),d0     ; v=*(short *)addr2
        move.w      d0,d1           ; vs=v
        add.w       d1,d1           ; vs<<=1
        move.w      d1,d2           ; v3=vs
```

Engineering:KlicsCode:CompPict:ConvolveSH3.a

```
        add.w       d0,d2           ; v3=vs+v
        sub.w       d2,&dAH         ; dAH-=v3
        add.w       d1,d1           ; vs<<=1
        add.w       d0,&dBG         ; dBG+=v
        add.w       d1,&dBG         ; dBG+=vs
        add.w       d2,&dAG         ; dAG+=v3
        add.w       d1,d1           ; vs<<=1
        add.w       d1,&dAG         ; dAG+=vs
        add.w       d2,&dBH         ; dBH+=v3
        add.w       d1,d1           ; vs<<=1
        add.w       d1,&dBH         ; dBH+=vs
        clr.w       d0              ; d0=0
        asr.w       #5,&dAH         ; dAH>>=5
        addx.w      d0,&dAH         ; round dAH
        asr.w       #5,&dAG         ; dAG>>=5
        addx.w      d0,&dAG         ; round dAG
        move.w      &dAH,(&addr0)   ; *(short *)addr0=dAH
        move.w      &dAG,(&addr1)   ; *(short *)addr1=dAG
*
        mend
*------------------------------------
        macro
        FwdEnd  &addr3,&addr2,&dBG,&dBH
*
        move.w      (&addr3),d0     ; v=*(short *)addr3
        add.w       d0,d0           ; v<<=1
        add.w       d0,&dBH         ; dBH+=v
        lsl.w       #2,d0           ; v<<=2
        sub.w       d0,&dBG         ; dBG-=v
        clr.w       d0              ; d0=0
        asr.w       #5,&dBH         ; dBH>>=5
        addx.w      d0,&dBH         ; round dBH
        asr.w       #5,&dBG         ; dBG>>=5
        addx.w      d0,&dBG         ; round dBG
        move.w      &dBH,(&addr2)   ; *(short *)addr2=dBH
        move.w      &dBG,(&addr3)   ; *(short *)addr3=dBG
*
        endm.
*------------------------------------
        macro
        Fwd         &base,&end,&inc
*
        movea.l     &base,a0                ; addr0=base
        move.l      &inc,d0                 ; d0=inc
        asr.l       #2,d0                   ; d0=inc>>2
        movea.l     a0,a3                   ; addr3=addr0
        suba.l      d0,a3                   ; addr3-=(inc>>2)
        movea.l     a3,a2                   ; addr2=addr3
        suba.l      d0,a2                   ; addr2-=(inc>>2)
        movea.l     a2,a1                   ; addr1=addr2
        suba.l      d0,a1                   ; addr1-=(inc>>2)
        FwdStart    a0,d4,d5                ; FwdStart(addr0,dAG,dAH)
        adda.l      &inc,a1                 ; addr1+=inc
        FwdOdd      a1,d4,d5,d6,d7          ; FwdOdd(addr1,dAG,dAH,dBG,dBH)
        adda.l      &inc,a2                 ; addr2+=inc
        FwdEven     a2,a1,a0,d4,d5,d6,d7    ; FwdEven(addr2,addr1,addr0,dAG,dAH,dB
        adda.l      &inc,a3                 ; addr3+=inc
@do     FwdOdd      a3,d6,d7,d4,d5          ; FwdOdd(addr3,dBG,dBH,dAG,dAH)
        adda.l      &inc,a0                 ; addr0+=inc
        FwdEven     a0,a3,a2,d6,d7,d4,d5    ; FwdEven(addr0,addr3,addr2,dBG,dBH,dA
        adda.l      &inc,a1                 ; addr1+=inc
        FwdOdd      a1,d4,d5,d6,d7          ; FwdOdd(addr1,dAG,dAH,dBG,dBH)
        adda.l      &inc,a2                 ; addr2+=inc
```

Engineering:KlicsCode:CompPict:ConvolveSH3.a

```
        FwdEven     a2,a1,a0,d4,d5,d6,d7      ; FwdEven(addr2,addr1,addr0,dAG,dAH,dB
        adda.l      &inc,a3                   ; addr3+=inc
        cmpa.l      a3,&end                   ; addr3<end
        bgt.w       @do                       ; while
        FwdEnd      a3,a2,d6,d7               ; FwdEnd(addr3,addr2,dBG,dBH)
*
        endm
*-----------------------------------
FastForward FUNC    EXPORT
*
        link        a6,#0                     ; no local variables
        movem.l     d4-d7/a3-a5,-(a7)         ; store registers
*
        move.l      $000C(a6),d3              ; inc=incl
        movea.l     $0008(a6),a5              ; base=data
@do     movea.l     a5,a4                     ; end=base
        adda.l      $0010(a6),a4              ; end+=endl
        Fwd         a5,a4,d3                  ; Fwd(base,end,inc)
        adda.l      $0014(a6),a5              ; base+=inc2
        cmpa.l      $0018(a6),a5              ; end2>base
        blt.w       @do                       ; for
*
        movem.l     (a7)+,d4-d7/a3-a5         ; restore registers
        unlk        a6                        ; remove locals
        rts                                   ; return
*
        ENDFUNC
*-----------------------------------
        macro
        BwdStart0   &addr0,&dAG,&dAH,&dBH
*
        move.w      (&addr0),d0       ; v=*(short *)addr0
        move.w      d0,d1             ; vs=v
        add.w       d1,d1             ; vs<<=1 (vs=2v)
        add.w       d1,d0             ; v+=vs (v=3v)
        move.w      d0,&dAG           ; dAG=v3
        neg.w       &dAG              ; dAG= -dAG
        move.w      d0,&dAH           ; dAH=v
        add.w       d1,&dAH           ; dAH+=vs
        lsl.w       #2,d1             ; vs<<=2 (vs=8v)
        move.w      d1,&dBH           ; dBH=vs
*
        endm
*-----------------------------------
        macro
        BwdStart1   &addr1,&addr0,&dAG,&dAH,&dBH
*
        move.w      (&addr1),d0       ; v=*(short *)addr1
        move.w      d0,d1             ; vs=v
        add.w       d1,d1             ; vs<<=1
        add.w       d1,&dBH           ; dBH+=vs
        add.w       d1,d0             ; v+=vs  (v=3v)
        lsl.l       #2,d1             ; vs<<=2 (vs=8v)
        add.w       d1,d0             ; v+=vs (v=11v)
        add.w       d0,&dAG           ; dAG+=v
        add.w       d1,d0             ; v+=vs (v=19v)
        sub.w       d0,&dAH           ; dAH-=v
*       clr.w       d0                ; d0=0
        asr.w       #3,&dBH           ; dBH>>=3
*       addx.w      d0,&dBH           ; round dBH
        move.w      &dBH,(&addr0)     ; *(short *)addr0=dBH
*
        endm
```

Engineering:KlicsCode:CompPict:ConvolveSH3.a

```
*-----------------------------------
        macro
        BwdEven &addr2,&dAG,&dAH,&dBG,&dBH
*
        move.w      (&addr2),d0         ; v=*(short *)addr2
        move.w      d0,d1               ; vs=v
        add.w       d1,d1               ; vs<<=1 (vs=2v)
        add.w       d1,d0               ; v+=vs  (v=3v)
        move.w      d0,&dBG             ; dBG=v
        neg.w       &dBG                ; dBG= -dBG
        move.w      d0,&dBH             ; dBH=v
        add.w       d1,&dBH             ; dBH+=vs
        lsl.w       #2,d1               ; vs<<=2 (vs=8v)
        add.w       d1,d0               ; v+=vs (v=11v)
        add.w       d0,&dAH             ; dAH+=v
        add.w       d1,d0               ; v+=vs (v=19v)
        add.w       d0,&dAG             ; dAG+=v
*
        endm
*-----------------------------------
        macro
        BwdOdd      &addr3,&addr2,&addr1,&dAG,&dAH,&dBG,&dBH
*
        move.w      (&addr3),d0         ; v=*(short *)addr3
        move.w      d0,d1               ; vs=v
        add.w       d1,d1               ; vs<<=1 (vs=2v)
        add.w       d1,d0               ; v+=vs  (v=3v)
        add.w       d0,&dAH             ; dAH+=v
        add.w       d0,&dAG             ; dAG+=v
        add.w       d1,&dAG             ; dAG+=vs
        lsl.w       #2,d1               ; vs<<=2 (vs=8v)
        add.w       d1,d0               ; v+=vs (v=11v)
        add.w       d0,&dBG             ; dBG+=v
        add.w       d1,d0               ; v+=vs (v=19v)
        sub.w       d0,&dBH             ; dBH-=v
        clr.w       d0                  ; d0=0
        asr.w       #4,&dAH             ; dAH>>=4
        addx.w      d0,&dAH             ; round dAH
        move.w      &dAH,(&addr1)       ; *(short *)addr1=dAH
        asr.w       #4,&dAG             ; dAG>>=4
        addx.w      d0,&dAG             ; round dAG
        move.w      &dAG,(&addr2)       ; *(short *)addr2=dAG
*
        endm
*-----------------------------------
        macro
        BwdEnd2     &addr2,&dAG,&dAH,&dBH
*
        move.w      (&addr2),d0         ; v=*(short *)addr2
        move.w      d0,d1               ; vs=v
        add.w       d1,d1               ; vs<<=1 (vs=2v)
        add.w       d1,d0               ; v+=vs  (v=3v)
        lsl.w       #2,d1               ; vs<<=2 (vs=8v)
        move.w      d1,&dBH             ; dBH=vs
        add.w       d1,d0               ; v+=vs (v=11v)
        add.w       d0,&dAH             ; dAH+=v
        add.w       d1,d0               ; v+=vs (v=19v)
        add.w       d0,&dAG             ; dAG+=v
*
        endm
*-----------------------------------
        macro
        BwdEnd3     &addr3,&addr2,&addr1,&dAG,&dAH,&dBH
```

Engineering:KlicsCode:CompPict:ConvolveSH3.a

```
        move.w      (&addr3),d0         ; v=*(short *)addr3
        move.w      d0,d1               ; vs=v
        add.w       d1,d1               ; vs<<=1 (vs=2v)
        add.w       d1,d0               ; v+=vs  (v=3v)
        add.w       d0,&dAH             ; dAH+=v
        add.w       d0,&dAG             ; dAG+=v
        add.w       d1,&dAG             ; dAG+=vs
        add.w       d1,&dBH             ; dBH+=vs
        lsl.l       #4,d1               ; vs<<=4  (v=32v)
        sub.w       d1,&dBH             ; dBH-=vs
        clr.w       d0                  ; d0=0
        asr.w       #4,&dAH             ; dAH>>=4
        addx.w      d0,&dAH             ; round dAH
        move.w      &dAH,(&addr1)       ; *(short *)addr1=dAH
        asr.w       #4,&dAG             ; dAG>>=4
        addx.w      d0,&dAG             ; round dAG
        move.w      &dAG,(&addr2)       ; *(short *)addr2=dAG
        asr.w       #3,&dBH             ; dBH>>=3
        addx.w      d0,&dBH             ; round dBH
        move.w      &dBH,(&addr3)       ; *(short *)addr3=dBH

        endm
*-------------------------------
        macro
        Bwd         &base,&end,&inc

        movea.l     &base,a0            ; addr0=base
        move.l      &inc,d0             ; d0=inc
        asr.l       #2,d0               ; d0=inc>>2
        movea.l     a0,a3               ; addr3=addr0
        suba.l      d0,a3               ; addr3-=(inc>>2)
        movea.l     a3,a2               ; addr2=addr3
        suba.l      d0,a2               ; addr2-=(inc>>2)
        movea.l     a2,a1               ; addr1=addr2
        suba.l      d0,a1               ; addr1-=(inc>>2)
        BwdStart0   a0,d4,d5,d7         ; BwdStart0(addr0,dAG,dAH,dBH)
        adda.l      &inc,a1             ; addr1+=inc
        BwdStart1   a1,a0,d4,d5,d7      ; BwdStart1(addr1,addr0,dAG,dAH,dBH)
        adda.l      &inc,a2             ; addr2+=inc
@do     BwdEven     a2,d4,d5,d6,d7      ; BwdEven(addr2,dAG,dAH,dBG,dBH)
        adda.l      &inc,a3             ; addr3+=inc
        BwdOdd      a3,a2,a1,d4,d5,d6,d7  ; BwdOdd(addr3,addr2,addr1,dAG,dAH,dBG
        adda.l      &inc,a0             ; addr0+=inc
        BwdEven     a0,d6,d7,d4,d5      ; BwdEven(addr0,dBG,dBH,dAG,dAH)
        adda.l      &inc,a1             ; addr1+=inc
        BwdOdd      a1,a0,a3,d6,d7,d4,d5  ; BwdOdd(addr1,addr0,addr3,dBG,dBH,dAG
        adda.l      &inc,a2             ; addr2+=inc
        cmpa.l      a2,&end             ; addr2<end
        bgt         @do                 ; while
        BwdEnd2     a2,d4,d5,d7         ; BwdEnd2(addr2,dAG,dAH,dBH)
        adda.l      &inc,a3             ; addr3+=inc
        BwdEnd3     a3,a2,a1,d4,d5,d7   ; BwdEnd3(addr3,addr2,addr1,dAG,dAH,dB

        endm
*-------------------------------
FastBackward    FUNC    EXPORT
*
        link        a6,#0               ; no local variables
        movem.l     d4-d7/a3-a5,-(a7)   ; store registers
*
        move.l      $000C(a6),d3        ; inc=incl
        movea.l     $0008(a6),a5        ; base=data
```

Engineering:KlicsCode:CompPict:ConvolveSH3.a

```
@do     movea.l   a5,a4                ; end=base
        adda.l    $0010(a6),a4         ; end+=end1
        Bwd       a5,a4,d3             ; Bwd(base,end,inc)
        adda.l    $0014(a6),a5         ; base+=inc2
        cmpa.l    $0018(a6),a5         ; end2>base
        blt.w     @do                  ; for
*
        movem.l   (a7)+,d4-d7/a3-a5    ; restore registers
        unlk      a6                   ; remove locals
        rts                            ; return
*
        ENDFUNC
*---------------------------------
        END
```

Engineering:KlicsCode:CompPict:Colour.c

```
/******************************
 *
 *  © Copyright 1993 KLICS Limited
 *  All rights reserved.
 *
 *  Written by: Adrian Lewis
 *
 ******************************/
/*
 *  Test versions of colour space conversions in C
 */

#include <Memory.h>
#include <QuickDraw.h>

#define NewPointer(ptr,type,size) \
    saveZone=GetZone(); \
    SetZone(SystemZone()); \
    if (nil==(ptr=(type)NewPtr(size))) { \
        SetZone(ApplicZone()); \
        if (nil==(ptr=(type)NewPtr(size))) { \
            SetZone(saveZone); \
            return(MemoryError()); \
        } \
    } \
    SetZone(saveZone);

typedef union {
    long    pixel;
    char    rgb[4];
} Pixel;

/* Special YUV space version */
#define rgb_yuv(pixmap,Yc) \
    pixel.pixel=0x808080^*pixmap++; \
    r=(short)pixel.rgb[1]; \
    g=(short)pixel.rgb[2]; g+=g; \
    b=(short)pixel.rgb[3]; \
    Y=(b<<3)-b; \
    g+=r; \
    Y+=g+g+g; \
    Y>>=4; \
    Y+=g; \
    *Yc++=Y; \
    Y>>=2; \
    U+=b-Y; \
    V+=r-Y;

#define limit(Y,low,high) \
    Y<(low<<2)?low<<2:Y>(high<<2)?high<<2:Y

/* Standard YUV space version - Bt294 CR07(0) mode limiting */
#define rgb_yuv32(pixmap,Yc) \
    pixel.pixel=0x808080^*pixmap++; \
    r=(long)pixel.rgb[1]; \
    g=(long)pixel.rgb[2]; \
    b=(long)pixel.rgb[3]; \
    Y= (306*r + 601*g + 117*b)>>8; \
    *Yc++ = limit(Y,16-128,235-128); \
    U+= (512*r - 429*g - 83*b)>>8; \
    V+= (-173*r - 339*g + 512*b)>>8;

void    RGB2YUV32(long *pixmap, short *Yc, short *Uc, short *Vc, int area, int wid
```

```
    long    *pixmap2=pixmap+cols, *row, *end=pixmap+area;
    short   *Yc2=Yc+width;

    while(pixmap<end) {
        row=pixmap+width;
        while(pixmap<row) {
            Pixel   pixel;
            long    r,g,b,Y,U=0,V=0;

            rgb_yuv32(pixmap,Yc);
            rgb_yuv32(pixmap,Yc);
            rgb_yuv32(pixmap2,Yc2);
            rgb_yuv32(pixmap2,Yc2);
            U>>=2;
            V>>=2;
            *Uc++=limit(U,16-128,240-128);
            *Vc++=limit(V,16-128,240-128);
        }
        pixmap+=cols+cols-width;
        pixmap2+=cols+cols-width;
        Yc+=width;
        Yc2+=width;
    }
}

typedef struct {
    short   ry, rv, by, bu;
} RGB_Tab;

OSErr RGBTable(long **tab)
{
    RGB_Tab *table;
    int     i;
    THz     saveZone;

    NewPointer(table,RGB_Tab*,256*sizeof(RGB_Tab));
    *tab=(long *)table;
    for(i=0;i<128;i++) {
        table[i].ry=306*i>>8;
        table[i].rv=173*i>>8;
        table[i].by=117*i>>8;
        table[i].bu=83*i>>8;
    }
    for(i=128;i<256;i++) {
        table[i].ry=306*(i-256)>>8;
        table[i].rv=173*(i-256)>>8;
        table[i].by=117*(i-256)>>8;
        table[i].bu=83*(i-256)>>8;
    }
    return(noErr);
}

typedef struct {
    short   ru, gu, bv, gv;
} UV32_Tab;

UV32_Tab *UV32_Table()
{
    UV32_Tab    *table;
    int     i;

    table=(UV32_Tab *)NewPtr(256*sizeof(UV32_Tab));
```

```
    for(i=0;i<128;i++) {
        table[i].ru=128+(1436*i>>10);
        table[i].gu=128+(-731*i>>10);
        table[i].bv=128+(1815*i>>10);
        table[i].gv=-352*i>>10;
    }
    for(i=128;i<256;i++) {
        table[i].ru=128+(1436*(i-256)>>10);
        table[i].gu=128+(-731*(i-256)>>10);
        table[i].bv=128+(1815*(i-256)>>10);
        table[i].gv=-352*(i-256)>>10;
    }
    return(table);
}

typedef struct {
    long    u, v;
} UV32Tab;

OSErr   UV32Table(long **tab)
{
    long    *ytab;
    UV32Tab *uvtab;
    int     i;
    THz     saveZone;

    NewPointer(*tab,long*,512*sizeof(long)+512*sizeof(UV32Tab));
    ytab=*tab;
    uvtab=(UV32Tab*)&ytab[512];
    for(i=-256;i<256;i++) {
        long    yyy, sp;

        sp=0x000000fe&(i<-128?0:i>127?255:i+128);
        yyy=sp; yyy<<=8;
        yyy|=sp; yyy<<=8;
        yyy|=sp;
        ytab[0x000001ff&i]=yyy;
    }
    for(i=-256;i<256;i++) {
        long    ru,gu,bv,gv;

        ru=0xfffffffe & (1436*i>>10);
        gu=0x000001fe & (-731*i>>10);
        bv=0x000001fe & (1815*i>>10);
        gv=0x000001fe & (-352*i>>10);

        uvtab[0x000001FF&i].u=((ru<<8)|gu)<<8;
        uvtab[0x000001FF&i].v=(gv<<8)|bv;
    }
    return(noErr);
}

typedef struct {
    short   u, v;
} UV16Tab;

OSErr   UV16Table(long **tab)
{
    short   *ytab;
    UV16Tab *uvtab;
    int     i;
    THz     saveZone;
```

```
    NewPointer(*tab,long*,512*sizeof(short)+512*sizeof(UV16Tab));
    ytab=*(short **)tab;
    uvtab=(UV16Tab*)&ytab[512];
    for(i=-256;i<256;i++) {
        long    yyy, sp;

        sp=0x0000001e&((i<-128?0:i>127?255:i+128)>>3);
        yyy=sp; yyy<<=5;
        yyy|=sp; yyy<<=5;
        yyy|=sp;
        ytab[0x000001ff&i]=yyy;
    }
    for(i=-256;i<256;i++) {
        long    ru,gu,bv,gv;

        ru=0xfffffffe & (1436*i>>13);
        gu=0x0000003e & (-731*i>>13);
        bv=0x0000003e & (1815*i>>13);
        gv=0x0000003e & (-352*i>>13);

        uvtab[0x000001FF&i].u=((ru<<5)|gu)<<5;
        uvtab[0x000001FF&i].v=(gv<<5)|bv;
    }
    return(noErr);
}

#define over(val) \
    ((0xFF00&(val)) == 0)?(char)val:val<0?0:255

/* Standard YUV space version */
#define yuv_rgb32(pixmap,Yc) \
    Y=(*Yc++)>>2; \
    pixel.rgb[1]=over(Y+r); \
    pixel.rgb[2]=over(Y+g); \
    pixel.rgb[3]=over(Y+b); \
    *pixmap++=pixel.pixel;


void    YUV2RGB32(long *pixmap, short *Yc, short *Uc, short *Vc, int area, int wid
{
    long    *pixmap2=pixmap+cols, *row, *end=pixmap+area;
    short   *Yc2=Yc+width;

    while(pixmap<end) {
        row=pixmap+width;
        while(pixmap<row) {
            Pixel   pixel;
            long    r,g,b,Y,U,V;

            U=(*Uc++)>>2;
            V=(*Vc++)>>2;
            r=128+(1436*U>>10);
            g=128+(-731*U - 352*V>>10);
            b=128+(1815*V>>10);

            yuv_rgb32(pixmap,Yc);
            yuv_rgb32(pixmap,Yc);
            yuv_rgb32(pixmap2,Yc2);
            yuv_rgb32(pixmap2,Yc2);
        }
        pixmap+=cols+cols-width;
        pixmap2+=cols+cols-width;
        Yc+=width;
```

```
        Yc2+=width;
    }
}

#define rgb32_yuv(pixmap,Yc) \
    pixel.pixel=0x808080^*pixmap++; \
    r=pixel.rgb[1]; \
    g=pixel.rgb[2]; \
    b=pixel.rgb[3]; \
    Y= (table[0xFF&r].ry + (g<<2)-table[0xFF&g].ry-table[0xFF&g].by + table[0xFF&b
    *Yc++ = limit(Y,16-128,235-128); \
    U+= (r<<1) -g -table[0xFF&g].rv - table[0xFF&b].bu; \
    V+= (b<<1) -g -table[0xFF&r].rv - table[0xFF&g].bu;

void    RGB32YUV(RGB_Tab *table,long *pixmap, short *Yc, short *Uc, short *Vc, int
{
    long    *pixmap2=pixmap+cols, *row, *end=pixmap+area;
    short   *Yc2=Yc+width;

    while(pixmap<end) {
        row=pixmap+width;
        while(pixmap<row) {
            Pixel   pixel;
            long    r,g,b,Y,U=0,V=0;

/*          rgb32_yuv(pixmap,Yc);*/
            pixel.pixel=0x808080^*pixmap++;
            r=pixel.rgb[1];
            g=pixel.rgb[2];
            b=pixel.rgb[3];
            Y= (table[0xFF&r].ry + (g<<2)-table[0xFF&g].ry-table[0xFF&g].by + tabl
            *Yc++ = limit(Y,16-128,235-128);
            U+= (r<<1) -g -table[0xFF&g].rv - table[0xFF&b].bu;
            V+= (b<<1) -g -table[0xFF&r].rv - table[0xFF&g].bu;

            rgb32_yuv(pixmap,Yc);
            rgb32_yuv(pixmap2,Yc2);
            rgb32_yuv(pixmap2,Yc2);
            U>>=2;
            V>>=2;
            *Uc++=limit(U,16-128,240-128);
            *Vc++=limit(V,16-128,240-128);
        }
        pixmap+=cols+cols-width;
        pixmap2+=cols+cols-width;
        Yc+=width;
        Yc2+=width;
    }
}

#define yuv_rgb32x2(pixmap,Y) \
    pixel.rgb[1]=over(Y+r); \
    pixel.rgb[2]=over(Y+g); \
    pixel.rgb[3]=over(Y+b); \
    pixmap[cols]=pixel.pixel; \
    *pixmap++=pixel.pixel;

void    YUV2RGB32x2(UV32_Tab *table,long *pixmap, short *Yc, short *Uc, short *Vc,
{
    long    *pixmap2=pixmap+2*cols, *row, *end=pixmap+area;
    short   *Yc2=Yc+width;
```

```
    while(pixmap<end) {
        long    Yold=*Yc>>2, Yold2=*Yc2>>2;

        row=pixmap+width*2;
        while(pixmap<row) {
            Pixel   pixel;
            long    r,g,b,Y,U,V;

            U=0x00FF&((*Uc++)>>2);
            V=0x00FF&((*Vc++)>>2);
            r=table[U].ru;
            g=table[U].gu+table[V].gv;
            b=table[V].bv;

            Y=(*Yc++)>>2;
            Yold=(Y+Yold)>>1;
            yuv_rgb32x2(pixmap,Yold);

            Yold=Y;
            yuv_rgb32x2(pixmap,Yold);

            Y=(*Yc++)>>2;
            Yold=(Y+Yold)>>1;
            yuv_rgb32x2(pixmap,Yold);

            Yold=Y;
            yuv_rgb32x2(pixmap,Yold);

            Y=(*Yc2++)>>2;
            Yold2=(Y+Yold2)>>1;
            yuv_rgb32x2(pixmap2,Yold2);

            Yold2=Y;
            yuv_rgb32x2(pixmap2,Yold2);

            Y=(*Yc2++)>>2;
            Yold2=(Y+Yold2)>>1;
            yuv_rgb32x2(pixmap2,Yold2);

            Yold2=Y;
            yuv_rgb32x2(pixmap2,Yold2);
        }
        pixmap+=4*cols-2*width;
        pixmap2+=4*cols-2*width;
        Yc+=width;
        Yc2+=width;
    }
}

#define yuv_rgb8(pixel,Yc,index,dith) \
    Y=*Yc++; \
    Y<<=3; \
    Y&= 0x3F00; \
    Y|= U; \
    pixel.rgb[index]=table[Y].rgb[dith];

void    YUV2RGB8(Pixel *table,long *pixmap, short *Yc, short *Uc, short *Vc, int a
{
    long    *pixmap2=pixmap+cols/4, *row, *end=pixmap+area/4;
    short   *Yc2=Yc+width;

    while(pixmap<end) {
```

Engineering:KlicsCode:CompPict:Colour.c

```
        row=pixmap+width/4;
        while(pixmap<row) {
            Pixel   pixel, pixel2;
            long    Y,U,V;

            U=*Uc++;
            V=*Vc++;
            U>>=2;
            V>>=6;
            U= (U&0xF0) | (V&0x0F);

            yuv_rgb8(pixel,Yc,0,3);
            yuv_rgb8(pixel,Yc,1,0);
            yuv_rgb8(pixel2,Yc2,0,1);
            yuv_rgb8(pixel2,Yc2,1,2);

            U=*Uc++;
            V=*Vc++;
            U>>=2;
            V>>=6;
            U= (U&0xF0) | (V&0x0F);

            yuv_rgb8(pixel,Yc,2,3);
            yuv_rgb8(pixel,Yc,3,0);
            yuv_rgb8(pixel2,Yc2,2,1);
            yuv_rgb8(pixel2,Yc2,3,2);

            *pixmap++=pixel.pixel;
            *pixmap2++=pixel2.pixel;
        }
        pixmap+=(cols+cols-width)/4;
        pixmap2+=(cols+cols-width)/4;
        Yc+=width;
        Yc2+=width;
    }
}

#define yuv_rgb8x2(pixel,pixel2,Y,index,dith,dith2) \
    Y&= 0x3F00; \
    Y|= U; \
    pixel.rgb[index]=table[Y].rgb[dith]; \
    pixel2.rgb[index]=table[Y].rgb[dith2];

void    YUV2RGB8x2(Pixel *table,long *pixmap, short *Yc, short *Uc, short *Vc, int
{
    long    *pixmap2=pixmap+cols/2, *row, *end=pixmap+area/4;
    short   *Yc2=Yc+width;

    while(pixmap<end) {
        long    Yold=*Yc<<3, Yold2=*Yc2<<3;

        row=pixmap+width/2;
        while(pixmap<row) {
            Pixel   pixel, pixel2, pixel3, pixel4;
            long    Y,U,V;

            U=*Uc++;
            V=*Vc++;
            U>>=2;
            V>>=6;
            U= (U&0x00F0) | (V&0x000F);

            Y=(*Yc++)<<3;
```

Engineering:KlicsCode:CompPict:Colour.c

```
            Yold=(Y+Yold)>>1;
            yuv_rgb8x2(pixel,pixel2,Y,0,3,1);

            Yold=Y;
            yuv_rgb8x2(pixel,pixel2,Y,1,0,2);
            Yold=Y;

            Y=(*Yc++)<<3;
            Yold=(Y+Yold)>>1;
            yuv_rgb8x2(pixel,pixel2,Y,2,3,1);

            Yold=Y;
            yuv_rgb8x2(pixel,pixel2,Y,3,0,2);
            Yold=Y;

            Y=(*Yc2++)<<3;
            Yold2=(Y+Yold2)>>1;
            yuv_rgb8x2(pixel3,pixel4,Y,0,3,1);

            Yold2=Y;
            yuv_rgb8x2(pixel3,pixel4,Y,1,0,2);
            Yold2=Y;

            Y=(*Yc2++)<<3;
            Yold2=(Y+Yold2)>>1;
            yuv_rgb8x2(pixel3,pixel4,Y,2,3,1);

            Yold2=Y;
            yuv_rgb8x2(pixel3,pixel4,Y,3,0,2);
            Yold2=Y;

            pixmap[cols/4]=pixel2.pixel;
            *pixmap++=pixel.pixel;

            pixmap2[cols/4]=pixel4.pixel;
            *pixmap2++=pixel3.pixel;
        }
        pixmap+=(cols+cols-width)/2;
        pixmap2+=(cols+cols-width)/2;
        Yc+=width;
        Yc2+=width;
    }
}

#define yuv_rgbTEST(pixel,index,Y) \
    rgb_col.red=(Y+r<<8); \
    rgb_col.green=(Y+g<<8); \
    rgb_col.blue=(Y+b<<8); \
    pixel.rgb[index]=Color2Index(&rgb_col);


void    YUV2RGBTEST(UV32_Tab *table,long *pixmap, short *Yc, short *Uc, short *Vc,
{
    long    *pixmap2=pixmap+cols/2, *row, *end=pixmap+area/4;
    short   *Yc2=Yc+width;

    while(pixmap<end) {
        long    Yold=*Yc<<3, Yold2=*Yc2<<3;

        row=pixmap+width/2;
        while(pixmap<row) {
            RGBColor    rgb_col;
            Pixel   pixel, pixel2;
```

Engineering:KlicsCode:CompPict:Colour.c

```
        long    r,g,b,Y,U,V;

        U=0x00FF&((*Uc++)>>2);
        V=0x00FF&((*Vc++)>>2);
        r=table[U].ru;
        g=table[U].gu+table[V].gv;
        b=table[V].bv;

        Y=(*Yc++)>>2;
        Yold=(Y+Yold)>>1;
        rgb_col.red=(Yold+r<<8);
        rgb_col.green=(Yold+g<<8);
        rgb_col.blue=(Yold+b<<8);
        pixel.rgb[0]=Color2Index(&rgb_col);

        Yold=Y;
        yuv_rgbTEST(pixel,1,Yold);

        Y=(*Yc++)>>2;
        Yold=(Y+Yold)>>1;
        yuv_rgbTEST(pixel,2,Yold);

        Yold=Y;
        yuv_rgbTEST(pixel,3,Yold);

        Y=(*Yc2++)>>2;
        Yold2=(Y+Yold2)>>1;
        yuv_rgbTEST(pixel2,0,Yold2);

        Yold2=Y;
        yuv_rgbTEST(pixel2,1,Yold2);

        Y=(*Yc2++)>>2;
        Yold2=(Y+Yold2)>>1;
        yuv_rgbTEST(pixel2,2,Yold2);

        Yold2=Y;
        yuv_rgbTEST(pixel2,3,Yold2);

        pixmap[cols/4]=pixel.pixel;
        *pixmap++=pixel.pixel;

        pixmap2[cols/4]=pixel2.pixel;
        *pixmap2++=pixel2.pixel;
    }
    pixmap+=(cols+cols-width)/2;
    pixmap2+=(cols+cols-width)/2;
    Yc+=width;
    Yc2+=width;
  }
}
```

Engineering:KlicsCode:CompPict:Colour.a

```
*--------------------------------
*
*   © Copyright 1993 KLICS Limited
*   All rights reserved.
*
*   Written by: Adrian Lewis
*
*--------------------------------
*
*   68030 Colour space conversions
*
*--------------------------------
        machine mc68030
        seg     'klics'
        include 'Traps.a'
*--------------------------------
        macro
        DPY32x2     &ARGB, &row, &o0, &o1, &n0, &n1
*
        add.l       &n0,&o0                 ;
        lsr.l       #1,&o0                  ; interpolate first pixel
        add.l       &n1,&o1                 ;
        lsr.l       #1,&o1                  ; interpolate first pixel

        move.l      &o0,(&ARGB)
        add.l       &row,&ARGB
        move.l      &o0,(&ARGB)
        add.l       &row,&ARGB
        move.l      &o1,(&ARGB)
        add.l       &row,&ARGB
        move.l      &o1,(&ARGB)+

        move.l      &n1,(&ARGB)
        sub.l       &row,&ARGB
        move.l      &n1,(&ARGB)
        sub.l       &row,&ARGB
        move.l      &n0,(&ARGB)
        sub.l       &row,&ARGB
        move.l      &n0,(&ARGB)+
*
        endm
*--------------------------------
        macro
        DPY32       &ARGB, &row, &o0, &o1, &n0, &n1
*
        move.l      &o0,(&ARGB)
        add.l       &row,&ARGB
        move.l      &o1,(&ARGB)+

        move.l      &n1,(&ARGB)
        sub.l       &row,&ARGB
        move.l      &n0,(&ARGB)+
*
        endm
*--------------------------------
        macro
        UV2RGB32    &AU,&AV,&TAB
*
        add.l       #2048,&TAB              ; move to uvtab

        move.w      &AU,d1                  ; Load U
        lsr.w       #2,d1
        and.w       #$01FF,d1
```

```
        move.l      (&TAB,d1.w*8),d0        ; UV now rg (u)

        move.w      &AV,d1                  ; Load V
        lsr.w       #2,d1
        and.w       #$01FF,d1
        add.l       4(&TAB,d1.w*8),d0       ; UV now rgb

        move.l      d0,d1                   ; 3 copies
        move.l      d0,d2
        move.l      d0,d3

        sub.l       #2048,&TAB              ; restore ytab
*
        endm
*-------------------------------
        macro
        GETY32      &AY, &TAB, &RGB0, &RGB1
*
        move.l      &AY,d4                  ;Y
        lsr.w       #2,d4
        and.w       #$01FF,d4
        add.l       (&TAB,d4.w*4),&RGB1     ; RGB1+=YYY

        swap        d4
        lsr.w       #2,d4
        and.w       #$01FF,d4
        add.l       (&TAB,d4.w*4),&RGB0     ; RGB0+=YYY
*
        endm
*-------------------------------
        macro
        OVER32      &RGB
*
        move.l      &RGB,d4             ; copy pixel
        andi.l      #$01010100,d4       ; was it this rgb
        beq.s       @nx_rgb             ; if not then quit
        btst        #24,d4              ; R overflow?
        beq.s       @bit16              ; if not then continue
        btst        #23,&RGB            ; test sign
        beq.s       @pos23              ; if positive
        andi.l      #$0000ffff,&RGB     ; underflow sets R to 0
        bra.s       @bit16              ; do next bit
@pos23  ori.l       #$00ff0000,&RGB     ; overflow sets R to 255
@bit16  btst        #16,d4              ; G overflow?
        beq.s       @bit8               ; if not then continue
        btst        #15,&RGB            ; test sign
        beq.s       @pos16              ; if positive
        andi.w      #$00ff,&RGB         ; underflow sets G to 0
        bra.s       @bit8               ; do next bit
@pos16  ori.w       #$ff00,&RGB         ; overflow sets G to 255
@bit8   btst        #8,d4               ; B overflow?
        beq.s       @end                ; if not then continue
        btst        #7,&RGB             ; test sign
        seq         &RGB                ; under/over flow
@end    andi.l      #$00fefefe,&RGB     ; mask RGB ok
@nx_rgb
*
        endm
*-------------------------------
        macro
        HASHOUT32   &AH, &D0, &D1, &D2, &D3
*
        move.l      &D0,d4
```

```
        add.l       &D1,d4
        add.l       &D2,d4
        add.l       &D3,d4
        andi.l      #$03e3e3e0,d4
        move.l      d4,&AH
*
        endm
*-------------------------------
        macro
        HASHCMP32   &AH, &D0, &D1, &D2, &D3
*
        move.l      &D0,d4
        add.l       &D1,d4
        add.l       &D2,d4
        add.l       &D3,d4
        andi.l      #$03e3e3e0,d4
        cmp.l       &AH,d4
*
        endm
*-------------------------------
OUT32X2 FUNC    EXPORT
*
PS      RECORD      8
table   DS.L        1
pixmap  DS.L        1
Y       DS.L        1
U       DS.L        1
V       DS.L        1
width   DS.L        1
height  DS.L        1
rowByte DS.L        1
pixmap2 DS.L        1
        ENDR
*
LS      RECORD      0,DECR
Y1      DS.L        1           ; sizeof(short)*Yrow            = 2*width
U_ex    DS.L        1           ; x end address                 = U+U_ix
U_ey    DS.L        1           ; y end address                 = U+width*height>>
U_ix    DS.L        1           ; sizeof(short)*UVrow           = width
Y_y     DS.L        1           ; sizeof(short)*Yrow            = 2*width
P_y     DS.L        1           ; 4*rowBytes-sizeof(long)*Prow  = 4*rowBytes-width
LSize   EQU         *
        ENDR
*
*       a0 - Y, a1 - U, a2 - V, a3 - pixmap, a4 - table, a5 - pixmap2
*       d0 - rgb00, d1 - rgb01, d2 - rgb10, d3 - rgb11, d4 - spare, d6 - old0, d7
*
        link        a6,#LS.LSize            ; inc, width, fend and rowend are loca
        movem.l     d4-d7/a3-a5,-(a7)       ; store registers
*
        move        SR,d0

        move.l      PS.Y(a6),a0             ; Y=Yc
        move.l      PS.U(a6),a1             ; U=Uc
        move.l      PS.V(a6),a2             ; V=Vc
        move.l      PS.pixmap(a6),a3        ; pm=pixmap
        move.l      PS.table(a6),a4         ; tab=table
        move.l      PS.pixmap2(a6),a5       ; pm2=pixmap2

        move.l      PS.width(a6),d0         ; LOAD width
        move.l      d0,LS.U_ix(a6)          ; SAVE U_ix
        move.l      PS.height(a6),d1        ; LOAD height
        mulu.w      d0,d1                   ; width*height
```

Engineering:KlicsCode:CompPict:Colour.a

```
        lsr.l       #1,d1                   ;  width*height/2
        add.l       a1,d1                   ;  U+width*height/2
        move.l      d1,LS.U_ey(a6)          ; SAVE U_ey
        add.l       d0,d0                   ;  width*2
        move.l      d0,LS.Y1(a6)            ; SAVE Y1
        move.l      d0,LS.Y_y(a6)           ; SAVE Y_y
        lsl.l       #2,d0                   ;  width*8
        move.l      PS.rowByte(a6),d1       ; LOAD rowBytes
        lsl.l       #2,d1                   ;  rowBytes*4
        sub.l       d0,d1                   ;  rowBytes*4-width*8
        move.l      d1,LS.P_y(a6)           ; SAVE P_y

        move.l      PS.rowByte(a6),d5       ; load rowBytes
        clr.l       d6                      ; clear old2
        clr.l       d7                      ; clear old1

@do_y   move.l      LS.U_ix(a6),d0          ; LOAD U_ixB
        add.l       a1,d0                   ;  P+U_ixB
        move.l      d0,LS.U_ex(a6)          ; SAVE U_exB

@do_x   UV2RGB32    (a1)+,(a2)+,a4          ; uv2rgb(*U++,*V++)

        move.l      LS.Y1(a6),d4            ; load Yrow
        GETY32      (a0,d4.l),a4,d2,d3      ; add Yb to RGB values
        GETY32      (a0)+,a4,d0,d1          ; add Ya to RGB values

        move.l      d0,d4
        or.l        d1,d4
        or.l        d2,d4
        or.l        d3,d4
        andi.l      #$01010100,d4
        bne.s       @over                   ; if overflow

@ok     HASHOUT32   (a5)+,d0,d1,d2,d3

        DPY32x2     a3,d5,d6,d7,d0,d2
        DPY32x2     a3,d5,d0,d2,d1,d3

        move.l      d1,d6                   ; copy olds
        move.l      d3,d7

        cmpa.l      LS.U_ex(a6),a1
        blt.w       @do_x

        add.l       LS.Y_y(a6),a0
        add.l       LS.P_y(a6),a3

        cmpa.l      LS.U_ey(a6),a1
        blt.w       @do_y

        movem.l     (a7)+,d4-d7/a3-a5       ; restore registers
        unlk        a6                      ; remove locals
        rts                                 ; return
@over   OVER32      d0
        OVER32      d1
        OVER32      d2
        OVER32      d3
        bra         @ok
*
        ENDFUNC
*--------------------------------
OUT32X2D    FUNC    EXPORT
*
```

Engineering:KlicsCode:CompPict:Colour.a

```
PS       RECORD    8
table    DS.L      1
pixmap   DS.L      1
Y        DS.L      1
U        DS.L      1
V        DS.L      1
width    DS.L      1
height   DS.L      1
rowByte  DS.L      1
pixmap2  DS.L      1
         ENDR
*
LS       RECORD    0,DECR
Y1       DS.L      1         ; sizeof(short)*Yrow              = 2*width
U_ex     DS.L      1         ; x end address                   = U+U_ix
U_ey     DS.L      1         ; y end address                   = U+width*height>>
U_ix     DS.L      1         ; sizeof(short)*UVrow             = width
Y_y      DS.L      1         ; sizeof(short)*Yrow              = 2*width
P_y      DS.L      1         ; 4*rowBytes-sizeof(long)*Prow    = 4*rowBytes-width
LSize    EQU       *
         ENDR
*
*        a0 - Y, a1 - U, a2 - V, a3 - pixmap, a4 - table, a5 - pixmap2
*        d0 - rgb00, d1 - rgb01, d2 - rgb10, d3 - rgb11, d4 - spare, d6 - old0, d7
*
         link      a6,#LS.LSize          ; inc, width, fend and rowend are loca
         movem.l   d4-d7/a3-a5,-(a7)     ; store registers
*
         move.l    PS.Y(a6),a0           ; Y=Yc
         move.l    PS.U(a6),a1           ; U=Uc
         move.l    PS.V(a6),a2           ; V=Vc
         move.l    PS.pixmap(a6),a3      ; pm=pixmap
         move.l    PS.table(a6),a4       ; tab=table
         move.l    PS.pixmap2(a6),a5     ; pm2=pixmap2

         move.l    PS.width(a6),d0       ; LOAD width
         move.l    d0,LS.U_ix(a6)        ; SAVE U_ix
         move.l    PS.height(a6),d1      ; LOAD height
         mulu.w    d0,d1                 ;  width*height
         lsr.l     #1,d1                 ;  width*hoight/2-
         add.l     a1,d1                 ;  U+width*height/2
         move.l    d1,LS.U_ey(a6)        ; SAVE U_ey
         add.l     d0,d0                 ;  width*2
         move.l    d0,LS.Y1(a6)          ; SAVE Y1
         move.l    d0,LS.Y_y(a6)         ; SAVE Y_y
         lsl.l     #2,d0                 ;  width*8
         move.l    PS.rowByre(a6),d1     ; LOAD rowbytes
         lsl.l     #2,d1                 ;  rowBytes*4
         sub.l     d0,d1                 ;  rowbytes*4-width*8
         move.l    d1,LS.P_y(a6)         ; SAVE P_y

         move.l    PS.rowByte(a6),d5     ; load rowBytes
         clr.l     d6                    ; clear old2
         clr.l     d7                    ; clear old1

@do_y    move.l    LS.U_ix(a6),d0        ; LOAD U_ixB
         add.l     a1,d0                 ;  P+U_ixB
         move.l    d0,LS.U_ex(a6)        ; SAVE U_exB

@do_x    UV2RGB32  (a1)+,(a2)+,a4        ; uv2rgb(*U++,*V++)

         move.l    LS.Y1(a6),d4          ; load Yrow
         GETY32    (a0,d4.1),a4,d2,d3    ; add Yb to RGB values
```

Engineering:KlicsCode:CompPict:Colour.a

```
        GETY32      (a0)+,a4,d0,d1          ; add Ya to RGB values

        move.l      d0,d4
        or.l        d1,d4
        or.l        d2,d4
        or.l        d3,d4
        andi.l      #$01010100,d4
        bne.w       @over                   ; if overflow

@ok     HASHCMP32   (a5)+,d0,d1,d2,d3
        bne.s       @diff

        add.l       #16,a3                  ; add four pixels

@cont   move.l      d1,d6                   ; copy olds
        move.l      d3,d7

        cmpa.l      LS.U_ex(a6),a1
        blt.w       @do_x

        add.l       LS.Y_y(a6),a0
        add.l       LS.P_y(a6),a3

        cmpa.l      LS.U_ey(a6),a1
        blt.w       @do_y
*
        movem.l     (a7)+,d4-d7/a3-a5       ; restore registers
        unlk        a6                      ; remove locals
        rts                                 ; return
@diff   move.l      d4,-4(a5)
        DPY32x2     a3,d5,d6,d7,d0,d2
        DPY32x2     a3,d5,d0,d2,d1,d3
        bra.s       @cont
@over   OVER32      d0
        OVER32      d1
        OVER32      d2
        OVER32      d3
        bra         @ok
*
        ENDFUNC
*---------------------------------
OUT32   FUNC    EXPORT
*
PS      RECORD      8
table   DS.L        1
pixmap  DS.L        1
Y       DS.L        1
U       DS.L        1
V       DS.L        1
width   DS.L        1
height  DS.L        1
rowByte DS.L        1
pixmap2 DS.L        1
        ENDR
*
LS      RECORD      0,DECR
Y1      DS.L        1       ; sizeof(short)*Yrow            = 2*width
U_ex    DS.L        1       ; x end address                 = U+U_ix
U_ey    DS.L        1       ; y end address                 = U+width*height>>
U_ix    DS.L        1       ; sizeof(short)*UVrow           = width
Y_y     DS.L        1       ; sizeof(short)*Yrow            = 2*width
P_y     DS.L        1       ; 2*rowBytes-sizeof(long)*Prow  = 2*rowBytes-width
LSize   EQU         *
```

Engineering:KiicsCode:CompPict:Colour.a

```
        ENDR
*
*       a0 - Y, a1 - U, a2 - V, a3 - pixmap, a4 - table, a5 - pixmap2
*       d0 - rgb00, d1 - rgb01, d2 - rgb10, d3 - rgb11, d4 - spare, d6 - cld0, d7
*
        link        a6,#LS.LSize            ; inc, width, fend and rowend are loca
        movem.l     d4-d7/a3-a5,-(a7)       ; store registers
*
        move.l      PS.Y(a6),a0             ; Y=Yc
        move.l      PS.U(a6),a1             ; U=Uc
        move.l      PS.V(a6),a2             ; V=Vc
        move.l      PS.pixmap(a6),a3        ; pm=pixmap
        move.l      PS.table(a6),a4         ; tab=table
        move.l      PS.pixmap2(a6),a5       ; pm2=pixmap2

        move.l      PS.width(a6),d0         ; LOAD width
        move.l      d0,LS.U_ix(a6)          ; SAVE U_ix
        move.l      PS.height(a6),d1        ; LOAD height
        mulu.w      d0,d1                   ;  width*height
        lsr.l       #1,d1                   ;  width*height/2
        add.l       a1,d1                   ;  U+width*height/2
        move.l      d1,LS.U_ey(a6)          ; SAVE U_ey
        add.l       d0,d0                   ;  width*2
        move.l      d0,LS.Y1(a6)            ; SAVE Y1
        move.l      d0,LS.Y_y(a6)           ; SAVE Y_y
        add.l       d0,d0                   ;  width*4
        move.l      PS.rowByte(a6),d1       ; LOAD rowBytes
        add.l       d1,d1                   ;  rowBytes*2
        sub.l       d0,d1                   ;  rowBytes*2-width*4
        move.l      d1,LS.P_y(a6)           ; SAVE P_y

        move.l      PS.rowByte(a6),d5       ; load rowBytes
        move.l      LS.Y1(a6),d6            ; load Yrow

@do_y   move.l      LS.U_ix(a6),d7          ; LOAD U_ixB
        add.l       a1,d7                   ;  P+U_ixB

@do_x   UV2RGB32    (a1)+,(a2)+,a4          ; uv2rgb(*U++,*V++)

        GETY32      (a0,d6.l),a4,d2,d3      ; add Yb to RGB values
        GETY32      (a0)+,a4,d0,d1          ; add Ya to RGB values

        move.l      d0,d4
        or.l        d1,d4
        or.l        d2,d4
        or.l        d3,d4
        andi.l      #$01010100,d4
        bne.s       @over                   ; if overflow

@ok     HASHOUT32   (a5)+,d0,d1,d2,d3

        DPY32       a3,d5,d0,d2,d1,d3

        cmpa.l      d7,a1
        blt.w       @do_x

        add.l       LS.Y_y(a6),a0
        add.l       LS.P_y(a6),a3

        cmpa.l      LS.U_ey(a6),a1
        blt.w       @do_y
*
        movem.l     (a7)+,d4-d7/a3-a5       ; restore registers
```

```
         unlk      a6                        ; remove locals
         rts                                 ; return
@over    OVER32    d0
         OVER32    d1
         OVER32    d2
         OVER32    d3
         bra       @ok
*
         ENDFUNC
*-----------------------------------
OUT32D   FUNC      EXPORT
*
PS       RECORD    8
table    DS.L      1
pixmap   DS.L      1
Y        DS.L      1
U        DS.L      1
V        DS.L      1
width    DS.L      1
height   DS.L      1
rowByte  DS.L      1
pixmap2  DS.L      1
         ENDR
*
LS       RECORD    0,DECR
Y1       DS.L      1       ; sizeof(short)*Yrow               = 2*width
U_ex     DS.L      1       ; x end address                    = U+U_ix
U_ey     DS.L      1       ; y end address                    = U+width*height>>
U_ix     DS.L      1       ; sizeof(short)*UVrow              = width
Y_y      DS.L      1       ; sizeof(short)*Yrow               = 2*width
P_y      DS.L      1       ; 2*rowBytes-sizeof(long)*Prow     = 2*rowBytes-width
LSize    EQU       *
         ENDR
*
*        a0 - Y, a1 - U, a2 - V, a3 - pixmap, a4 - table, a5 - pixmap2
*        d0 - rgb00, d1 - rgb01, d2 - rgb10, d3 - rgb11, d4 - spare, d6 - Yrow, d7
*
         link      a6,#LS.LSize              ; inc, width, fend and rowend are loca
         movem.l   d4-d7/a3-a5,-(a7)         ; store registers
*
         move.l    PS.Y(a6),a0               ; Y=Yc
         move.l    PS.U(a6),a1               ; U=Uc
         move.l    PS.V(a6),a2               ; V=Vc
         move.l    PS.pixmap(a6),a3          ; pm=pixmap
         move.l    PS.table(a6),a4           ; tab=table
         move.l    PS.pixmap2(a6),a5         ; pm2=pixmap2

         move.l    PS.width(a6),d0           ; LOAD width
         move.l    d0,LS.U_ix(a6)            ; SAVE U_ix
         move.l    PS.height(a6),d1          ; LOAD height
         mulu.w    d0,d1                     ;  width*height
         lsr.l     #1,d1                     ;  width*height/2
         add.l     a1,d1                     ;  U+width*height/2
         move.l    d1,LS.U_ey(a6)            ; SAVE U_ey
         add.l     d0,d0                     ;  width*2
         move.l    d0,LS.Y1(a6)              ; SAVE Y1
         move.l    d0,LS.Y_y(a6)             ; SAVE Y_y
         add.l     d0,d0                     ;  width*4
         move.l    PS.rowByte(a6),d1         ; LOAD rowBytes
         add.l     d1,d1                     ;  rowBytes*2
         sub.l     d0,d1                     ;  rowBytes*2-width*4
         move.l    d1,LS.P_y(a6)             ; SAVE P_y
```

```
        move.l      PS.rowByte(a6),d5           ; load rowBytes
        move.l      LS.Y1(a6),d6                ; load Yrow

@do_y   move.l      LS.U_ix(a6),d7              ; LOAD U_ixB
        add.l       a1,d7                       ;  P+U_ixB

@do_x   UV2RGB32    (a1)+,(a2)+,a4              ; uv2rgb(*U++,*V++)

        move.l      LS.Y1(a6),d4                ; load Yrow
        GETY32      (a0,d6.l),a4,d2,d3          ; add Yb to RGB values
        GETY32      (a0)+,a4,d0,d1              ; add Ya to RGB values

        move.l      d0,d4
        or.l        d1,d4
        or.l        d2,d4
        or.l        d3,d4
        andi.l      #$01010100,d4
        bne.s       @over                       ; if overflow

@ok     HASHCMP32   (a5)+,d0,d1,d2,d3
        bne.s       @diff

        addq        #8,a3                       ; add four pixels

@cont   cmpa.l      d7,a1
        blt.w       @do_x

        add.l       LS.Y_y(a6),a0
        add.l       LS.P_y(a6),a3

        cmpa.l      LS.U_ey(a6),a1
        blt.w       @do_y
*
        movem.l     (a7)+,d4-d7/a3-a5           ; restore registers
        unlk        a6                          ; remove locals
        rts                                     ; return
@diff   move.l      d4,-4(a5)
        DPY32       a3,d5,d0,d2,d1,d3
        bra.s       @cont
@over   OVER32      d0
        OVER32      d1
        OVER32      d2
        OVER32      d3
        bra         @ok
*
        ENDFUNC
*---------------------------------
        macro
        UVOV        &VAL, &OV
*
        move.w      &VAL,&OV
        add.w       #$0200,&OV
        and.w       #$FC00,&OV
        beq.s       @ok
        tst.w       &OV
        bge.s       @pos
        move.w      #$01FF,&VAL
        bra.s       @ok
@pos    move.w      #$FE00,&VAL
@ok
*
        endm
```

Engineering:KlicsCode:CompPict:Colour.a

```
UVLIMIT FUNC        EXPORT
* fix d0, d4, spare d1 d2
        UVOV        d0,d1
        swap        d0
        UVOV        d0,d1
        swap        d0
        UVOV        d4,d1
        swap        d4
        UVOV        d4,d1
        swap        d4
        rts
*
        ENDFUNC

        macro
        UVOVER      &U, &V
*
        move.l      #$02000200,d1
        move.l      d1,d2
        add.l       &U,d1
        add.l       &V,d2
        or.l        d2,d1
        andi.l      #$FC00FC00,d1
        beq.s       @UVok
        bsr         UVLIMIT
@UVok
*
        endm

        macro
        GETUV       &AU, &AV, &SP, &UV
*
        move.l      (&AU)+,&SP
        move.l      (&AV)+,&UV
        UVOVER      &SP,&UV
        lsr.l       #5,&UV
        andi.l      #$03e003e0,&SP
        andi.l      #$001F001F,&UV
        or.l        &SP,&UV                     ; UV==$00UV00UV
        swap        &UV
*
        endm

        macro
        GETY        &AY,&IND,&UV,&R0,&R1
*
        move.l      &AY,&R1                     ; (2+) Y=Y0Y1
        lsl.l       #5,&R1                      ; (4) Y=Y0XXY1XX
        andi.l      #$FC00FC00,&R1              ;
        or.w        &UV,&R1                     ; (2) Y=Y1UV
        move.l      (&IND,&R1 .w*4),&R0         ; (2+) R0=0123 (Y1)
        swap        &R1                         ; (4) Y=Y0XX
        or.w        &UV,&R1                     ; (2) Y=Y0UV
        move.l      (&IND,&R1 .w*4),&R1         ; (2+) R1=0123 (Y0)
*
        endm

        macro
        UV8         &AU, &AV, &SP, &UV
*
        move.l      (&AU)+,&SP
        move.l      (&AV)+,&UV
        UVOVER      &SP,&UV
```

Engineering:KlicsCode:CompPict:Colour.a

```
        lsr.l       #2,&SP
        lsr.l       #6,&UV
        andi.l      #$00F000F0,&SP
        andi.l      #$000F000F,&UV
        or.l        &SP,&UV                 ; UV==$00UV00UV
        swap        &UV
*
        endm

        macro
        Y2IND       &Y,&IND,&UV,&D0,&D1
*
        move.l      &Y,&D0                  ; d0=Y0Y1
        lsl.l       #3,&D0                  ; d0=Y0XXY1XX
        move.b      &UV,&D0                 ; d0=Y0XXY1UV
        andi.w      #$3FFF,&D0              ; d0=0YUV(1)
        move.l      (&IND,&D0 .w*4),&D1     ; find clut entries
        swap        &D0                     ; d0=Y0XX
        move.b      &UV,&D0                 ; d0=Y0UV
        andi.w      #$3FFF,&D0              ; d0=0YUV(0)
        move.l      (&IND,&D0 .w*4),&D0     ; find clut entries
*
        endm

OUT8    FUNC    EXPORT
*
PS      RECORD      8
table   DS.L        1
pixmap  DS.L        1
Y       DS.L        1
U       DS.L        1
V       DS.L        1
width   DS.L        1
height  DS.L        1
rowByte DS.L        1
pixmap2 DS.L        1
        ENDR
*
LS      RECORD      0,DECR
Y1      DS.L        1       ; sizeof(short)*Yrow            = 2*width
U_ex    DS.L        1       ; x end address                 = U+U_ix
U_ey    DS.L        1       ; y end address                 = U+width*height>>
U_ix    DS.L        1       ; sizeof(short)*UVrow           = width
Y_y     DS.L        1       ; sizeof(short)*Yrow            = 2*width
P_y     DS.L        1       ; 2*rowBytes-sizeof(long)*Prow  = 2*rowBytes-width
LSize   EQU         *
        ENDR
*
*       a0 - Y, a1 - U, a2 - V, a3 - pixmap, a4 - table, a5 - pixmap2
*       d0 - rgb00, d1 - rgb01, d2 - rgb10, d3 - rgb11, d4 - spare, d6 - old0, d7
*
        link        a6,#LS.LSize            ; inc, width, fend and rowend are loca
        movem.l     d4-d7/a3-a5,-(a7)       ; store registers
*
        move.l      PS.Y(a6),a0             ; Y=Yc
        move.l      PS.U(a6),a1             ; U=Uc
        move.l      PS.V(a6),a2             ; V=Vc
        move.l      PS.pixmap(a6),a3        ; pm=pixmap
        move.l      PS.table(a6),a4         ; tab=table
        adda.l      #$00020000,a4           ; tab+=32768 (longs)
        move.l      PS.pixmap2(a6),a5       ; pm2=pixmap2

        move.l      PS.width(a6),d0         ; LOAD width
```

Engineering:KlicsCode:CompPict:Colour.a

```
        move.l     d0,LS.U_ix(a6)            ; SAVE U_ix
        move.l     PS.height(a6),d1          ; LOAD height
        mulu.w     d0,d1                     ;  width*height
        lsr.l      #1,d1                     ;  width*height/2
        add.l      a1,d1                     ;  U+width*height/2
        move.l     d1,LS.U_ey(a6)            ; SAVE U_ey
        move.l     PS.rowByte(a6),d1         ; LOAD rowBytes
        add.l      d1,d1                     ;  rowBytes*2
        sub.l      d0,d1                     ;  rowBytes*2-width
        move.l     d1,LS.P_y(a6)             ; SAVE P_y
        add.l      d0,d0                     ;  width*2
        move.l     d0,LS.Y1(a6)              ; SAVE Y1
        move.l     d0,LS.Y_y(a6)             ; SAVE Y_y

        move.l     PS.rowByte(a6),d5         ; load rowBytes
        move.l     LS.Y1(a6),d6              ; load Yrow

@do_y   move.l     LS.U_ix(a6),d7            ; LOAD U_ixB
        add.l      a1,d7                     ;  P+U_ixB

@do_x   GETUV      a1,a2,d0,d4

        GETY       (a0,d6.w),a4,d4,d2,d3     ; d2=X0XX. d3=XX1X
        GETY       (a0)+,a4,d4,d0,d1         ; d0=XXX0, d1=1XXX

        move.w     d3,d2                     ; d2=X01X
        lsl.l      #8,d2                     ; d2=01XX
        move.w     d0,d1                     ; d1=1XX0
        swap       d1                        ; d1=X01X
        lsl.l      #8,d1                     ; d1=01XX

        swap       d4                        ; next UV

        GETY       (a0,d6.l),a4,d4,d0,d3     ; d0=X2XX, d3=XX3X
        move.w     d3,d0                     ; d0=X23X
        lsr.l      #8,d0                     ; d0=XX23
        move.w     d0,d2                     ; d2=0123
        GETY       (a0)+,a4,d4,d0,d3         ; d0=XXX2, d3=3XXX
        move.w     d0,d3                     ; d3=3XX2
        swap       d3                        ; d3=X23X
        lsr.l      #8,d3                     ; d3=XX23
        move.w     d3,d1                     ; d1=0123

        move.l     d2,(a3,d5)
        move.l     d1,(a3)+

        cmpa.l     d7,a1
        blt.w      @do_x

        add.l      LS.Y_y(a6),a0
        add.l      LS.P_y(a6),a3

        cmpa.l     LS.U_ey(a6),a1
        blt.w      @do_y
*
        movem.l    (a7)+,d4-d7/a3-a5         ; restore registers
        unlk       a6                        ; remove locals
        rts        ; return
*
        ENDFUNC
*--------------------------------
        macro
        Y8x2       &AY,&IND,&UV,&old
```

```
move.l    &AY.d0                      ; (2+) Y=Y0Y1
lsl.l     #3,d0                       ; (4) Y=Y0XXY1XX
swap      d0                          ; (4) Y=Y1XXY0XX
add.w     d0,&old                     ; (2) old=old+Y0
lsr.w     #1,&old                     ; (4) old=(old+Y0)/2
move.b    &UV,&old                    ; (2) old=YI0UV
andi.w    #S3FFF,&old                 ; (4) old=0YUV(I0)
move.l    (&IND,&old .w*4),d1         ; (2+) d1=X1X3
move.w    d0,&old                     ; (2) old=Y0
move.b    &UV,d0                      ; (2) Y=Y0UV
andi.w    #S3FFF,d0                   ; (4) Y=0YUV(0)
move.l    (&IND,d0.w*4),d2            ; (2+) d2=0X2X
move.w    d1,d3                       ; (2) exg.w d1,d2
move.w    d2,d1                       ; (2) d1=X12X
move.w    d3,d2                       ; (2) d2=0XX3
swap      d2                          ; (4) d2=X30X
lsl.l     #8,d1                       ; (4) d1=12XX
lsl.l     #8,d2                       ; (4) d2=30XX
swap      d0                          ; (4) Y=Y1XX
add.w     d0,&old                     ; (2) old=old+Y1
lsr.w     #1,&old                     ; (4) old=(old+Y1)/2
move.b    &UV,&old                    ; (2) old=YI1UV
andi.w    #S3FFF,&old                 ; (4) old=0YUV(I1)
move.l    (&IND,&old .w*4),d3         ; (2+) d3=X1X3
move.w    d0,&old                     ; (2) old=Y1
move.b    &UV,d0                      ; (2) Y=Y0UV
andi.w    #S3FFF,d0                   ; (4) Y=0YUV(0)
move.l    (&IND,d0.w*4),d0            ; (2+) d0=0X2X
move.w    d0,d1                       ; (2) exg.w d0,d3
move.w    d3,d0                       ; (2) d0=0XX3
move.w    d1,d3                       ; (2) d3=X12X
swap      d0                          ; (4) d0=X30X
lsr.l     #8,d0                       ; (4) d0=XX30
lsr.l     #8,d3                       ; (4) d3=X12X
move.w    d0,d2                       ; (2) d2=3030 (YiY0YiY1) (1)
move.w    d3,d1                       ; (2) d1=2121 (YiY0YiY1) (2)

endm
macro
Y8x2a     &AY,&IND,&UV

GETY      &AY,&IND,&UV,d1,d2
move.l    &AY,d2                      ; (2+) Y=Y0Y1
lsl.l     #3,d2                       ; (4) Y=Y0XXY1XX
move.b    &UV,d2                      ; (2) Y=Y1UV
andi.w    #S3FFF,d2                   ; (4) Y=0YUV(Y1)
move.l    (&IND,d2.w*4),d1            ; (2+) d1=0123 (Y1)
swap      d2                          ; (4) Y=Y0XX
move.b    &UV,d2                      ; (2) Y=Y0UV
andi.w    #S3FFF,d2                   ; (4) Y=0YUV(Y0)
move.l    (&IND,d2.w*4),d2            ; (2+) d2=0123 (Y0)
move.w    d1,d0                       ; (2) exg.w d2,d1
move.w    d2,d1                       ; (2) d1=0123 (Y1Y0)
move.w    d0,d2                       ; (2) d2=0123 (Y0Y1)
swap      d1                          ; (4) d1=2301 (Y0Y1)

endm

macro
Y8x2b     &AY,&IND,&UV

GETY      &AY,&IND,&UV,d1,d2
```

Engineering:KlicsCode:CompPict:Colour.a

```
*        move.l     &AY,d2                  ; (2+) Y=Y0Y1
*        lsl.l      #3,d2                   ; (4) Y=Y0XXY1XX
*        move.b     &UV,d2                  ; (2) Y=Y1UV
*        andi.w     #$3FFF,d2               ; (4) Y=0YUV(Y1)
*        move.l     (&IND,d2.w*4),d1        ; (2+) d1=0123 (Y1)
*        swap       d2                      ; (4) Y=Y0XX
*        move.b     &UV,d2                  ; (2) Y=Y0UV
*        andi.w     #$3FFF,d2               ; (4) Y=0YUV(Y0)
*        move.l     (&IND,d2.w*4),d2        ; (2+) d2=0123 (Y0)
         ror.l      #8,d2                   ; (6) d2=3012 (Y0)
         ror.l      #8,d1                   ; (6) d1=3012 (Y1)
         move.w     d1,d0                   ; (2) exg.w d2,d1
         move.w     d2,d1                   ; (2) d1=3012 (Y1Y0)
         move.w     d0,d2                   ; (2) d2=3012 (Y0Y1)
         swap       d1                      ; (4) d1=1230 (Y0Y1)
         ror.w      #8,d1                   ; (6) d1=1203 (Y0Y1)
*
         endm

OUT8x2   FUNC       EXPORT
*
PS       RECORD     8
table    DS.L       1
pixmap   DS.L       1
Y        DS.L       1
U        DS.L       1
V        DS.L       1
width    DS.L       1
height   DS.L       1
rowByte  DS.L       1
pixmap2  DS.L       1
         ENDR
*
LS       RECORD     0,DECR
Y1       DS.L       1        ; sizeof(short)*Yrow              = 2*width
U_ex     DS.L       1        ; x end address                   = U+U_ix
U_ey     DS.L       1        ; y end address                   = U+width*height>>
U_ix     DS.L       1        ; sizeof(short)*UVrow             = width
Y_y      DS.L       1        ; sizeof(short)*Yrow              = 2*width
P_y      DS.L       1        ; 4*rowBytes-sizeof(long)*Prow    = 4*rowBytes-width
LSize    EQU        *
         ENDR
*
*        a0 - Y, a1 - U, a2 - V, a3 - pixmap, a4 - table, a5 - pixmap2
*        d0 - rgb00, d1 - rgb01, d2 - rgb10, d3 - rgb11, d4 - spare, d5 - old0, d7
*
         link       a6,#LS.LSize            ; inc, width, fend and rowend are loca
         movem.l    d4-d7/a3-a5,-(a7)       ; store registers
*
         move.l     PS.Y(a6),a0             ; Y=Yc
         move.l     PS.U(a6),a1             ; U=Uc
         move.l     PS.V(a6),a2             ; V=Vc
         move.l     PS.pixmap(a6),a3        ; pm=pixmap
         move.l     PS.table(a6),a4         ; tab=table
         adda.l     #$00020000,a4           ; tab+=32768 (longs)
         move.l     PS.pixmap2(a6),a5       ; pm2=pixmap2

         move.l     PS.width(a6),d0         ; LOAD width
         move.l     d0,LS.U_ix(a6)          ; SAVE U_ix
         move.l     PS.height(a6),d1        ; LOAD height
         mulu.w     d0,d1                   ;  width*height
         lsr.l      #1,d1                   ;  width*height/2
```

```
        add.l     a1,d1                    ;  U+width*height/2
        move.l    d1,LS.U_ey(a6)           ; SAVE U_ey
        add.l     d0,d0                    ;  width*2
        move.l    d0,LS.Y1(a6)             ; SAVE Y1
        move.l    d0,LS.Y_y(a6)            ; SAVE Y_y
        move.l    PS.rowByte(a6),d1        ; LOAD rowBytes
        add.l     d1,d1                    ;  rowBytes*2
        add.l     d1,d1                    ;  rowBytes*4
        sub.l     d0,d1                    ;  rowBytes*4-width*2
        move.l    d1,LS.P_y(a6)            ; SAVE P_y

        move.l    PS.rowByte(a6),d5        ; load rowBytes
        clr.l     d6
        clr.l     d7

@do_y   move.l    LS.U_ix(a6),d0           ; LOAD U_ixB
        add.l     a1,d0                    ;  P+U_ixB
        move.l    d0,LS.U_ex(a6)           ; SAVE U_exB

@do_x   GETUV     a1,a2,d0,d4              ; d4=00UV00UV (10)

        Y8x2a     (a0),a4,d4;,d6           ; calc d2,d1 pixels
        move.l    d2,(a3)
        add.l     d5,a3
        move.l    d1,(a3)
        add.l     d5,a3

        move.l    LS.Y1(a6),d0             ; load Yrow
        Y8x2b     (a0,d0.w),a4,d4;,d7      ; calc d2,d1 pixels
        move.l    d2,(a3)
        add.l     d5,a3
        move.l    d1,(a3)+

        swap      d4                       ; next UV
        addq.l    #4,a0                    ; next Ys

        move.l    LS.Y1(a6),d0             ; load Yrow
        Y8x2b     (a0,d0.w),a4,d4;,d7      ; calc d2,d1 pixels
        move.l    d1,(a3)
        sub.l     d5,a3
        move.l    d2,(a3)
        sub.l     d5,a3

        Y8x2a     (a0)+,a4,d4;,d6
        move.l    d1,(a3)
        sub.l     d5,a3
        move.l    d2,(a3)+

        cmpa.l    LS.U_ex(a6),a1
        blt.w     @do_x

        add.l     LS.Y_y(a6),a0
        add.l     LS.P_y(a6),a3

        cmpa.l    LS.U_ey(a6),a1
        blt.w     @do_y

        movem.l   (a7)+,d4-d7/a3-a5        ; restore registers
        unlk      a6                       ; remove locals
        rts       ; return

        ENDFUNC
;-------------------------------
```

Engineering:KlicsCode:CompPict:Colour.a

```
        macro
        RGB2Y         &RGB,&Y,&U,&V,&AY
*
        move.l        &RGB,d2                   ; pixel=*pixmap
        eori.l        #$808080,d2               ; pixel^=0x808080
        clr.w         d1                        ; B=0
        move.b        d2,d1                     ; B=pixel[3]
        move.l        4(a4,d1.w*8),d0           ; d0=by,bu
        sub.w         d0,&U                     ; U-=bu
        swap          d0                        ; d0=bu,by
        move.w        d0,&Y                     ; Y=by
        ext.w         d1                        ; (short)B
        add.w         d1,d1                     ; B*=2
        add.w         d1,&V                     ; V+=B<<1
        lsr.l         #8,d2                     ; pixel>>=8
        clr.w         d1                        ; G=0
        move.b        d2,d1                     ; G=pixel[3]
        move.l        (a4,d1.w*8),d0            ; d0=gry,gv
        sub.w         d0,&U                     ; U-=gv
        swap          d0                        ; d0=gv,gry
        sub.w         d0,&Y                     ; Y-=gry
        move.l        4(a4,d1.w*8),d0           ; d0=gby,gu
        sub.w         d0,&V                     ; V-=gv
        swap          d0                        ; d0=gu,gby
        sub.w         d0,&Y                     ; Y-=gby
        ext.w         d1                        ; (short)G
        sub.w         d1,&U                     ; U-=g
        sub.w         d1,&V                     ; V-=g
        lsl.w         #2,d1                     ; G<<=2
        add.w         d1,&Y                     ; Y+=B<<1
        lsr.l         #8,d2                     ; pixel>>=8
        move.l        (a4,d2.w*8),d0            ; d0=ry,rv
        sub.w         d0,&V                     ; V-=rv
        swap          d0                        ; d0=rv,ry
        add.w         d0,&Y                     ; Y+=ry
        ext.w         d2                        ; (short)R
        add.w         d2,d2                     ; R*=2
        add.w         d2,&U                     ; U+=R<<2
        cmpi.w        #$FE40,&Y                 ; Y>=-448
        bge.s         @ok                       ; if greater
        move.w        #$FE40,&Y                 ; Y= -448
        bra.s         @end                      ; save
@ok     cmpi.w        #$01C0,&Y                 ; Y< 448
        blt.s         @end                      ; if less
        move.w        #$01C0,&Y                 ; Y= 448
@end    move.w        &Y,&AY                    ; Save Y
*
        endm

IN32    FUNC      EXPORT
*
PS      RECORD         8
table   DS.L           1
pixmap  DS.L           1
Y       DS.L           1
U       DS.L           1
V       DS.L           1
width   DS.L           1
height  DS.L           1
rowByte DS.L           1
        ENDR
*
LS      RECORD         0,DECR
```

Engineering:KlicsCode:CompPict:Colour.a

```
Y1      DS.L       1          ; sizeof(short)*Yrow           = 2*width
U_ex    DS.L       1          ; x end address                = U+U_ix
U_ey    DS.L       1          ; y end address                = U+width*height>>
U_ix    DS.L       1          ; sizeof(short)*UVrow          = width
Y_y     DS.L       1          ; sizeof(short)*Yrow           = 2*width
P_y     DS.L       1          ; 2*rowBytes-sizeof(long)*Prow = 2*rowBytes-width
LSize   EQU        *
        ENDR
*
*       a0 - Y, a1 - U, a2 - V, a3 - pixmap, a4 - table, a5 - pixmap2
*       d0 - rgb00, d1 - rgb01, d2 - rgb10, d3 - rgb11, d4 - spare, d6 - old0, d7
*
        link       a6,#LS.LSize           ; inc, width, fend and rowend are loca
        movem.l    d4-d7/a3-a5,-(a7)      ; store registers
*
        move.l     PS.Y(a6),a0            ; Y=Yc
        move.l     PS.U(a6),a1            ; U=Uc
        move.l     PS.V(a6),a2            ; V=Vc
        move.l     PS.pixmap(a6),a3       ; pm=pixmap
        move.l     PS.table(a6),a4        ; tab=table

        move.l     PS.width(a6),d0        ; LOAD width
        move.l     d0,LS.U_ix(a6)         ; SAVE U_ix
        move.l     PS.height(a6),d1       ; LOAD height
        mulu.w     d0,d1                  ;  width*height
        lsr.l      #1,d1                  ;  width*height/2
        add.l      a1,d1                  ;  U+width*height/2
        move.l     d1,LS.U_ey(a6)         ; SAVE U_ey
        add.l      d0,d0                  ;  width*2
        move.l     d0,LS.Y1(a6)           ; SAVE Y1
        move.l     d0,LS.Y_y(a6)          ; SAVE Y_y
        add.l      d0,d0                  ;  width*4
        move.l     PS.rowByte(a6),d1      ; LOAD rowBytes
        add.l      d1,d1                  ;  rowBytes*2
        sub.l      d0,d1                  ;  rowBytes*2-width*4
        move.l     d1,LS.P_y(a6)          ; SAVE P_y

        move.l     PS.rowByte(a6),d7      ; load rowBytes
        move.l     LS.Y1(a6),d6           ; load Y1

@do_y   move.l     LS.U_ix(a6),d0         ; LOAD U_ixB
        add.l      a1,d0                  ;  P+U_ixB
        move.l     d0,LS.U_ex(a6)         ; SAVE U_exB

@do_x   clr.w      d4                     ; U=0
        clr.w      d5                     ; V=0

        RGB2Y      (a3,d7.w),d3,d4,d5,(a0,d6.w); Convert pixel
        RGB2Y      (a3)+,d3,d4,d5,(a0)+        ; Convert pixel
        RGB2Y      (a3,d7.w),d3,d4,d5,(a0,d6.w); Convert pixel
        RGB2Y      (a3)+,d3,d4,d5,(a0)+        ; Convert pixel

        asr.w      #2,d4                  ; U>>=2
        asr.w      #2,d5                  ; V>>=2

        cmpi.w     #$FE40,d4              ; U>=-448
        bge.s      @okU                   ; if greater
        move.w     #$FE40,d4              ; U= -448
        bra.s      @doV                   ; save
@okU    cmpi.w     #$01C0,d4              ; U< 448
        blt.s      @doV                   ; if less
        move.w     #$01C0,d4              ; U= 448
```

Engineering:KlicsCode:CompPict:Colour.a

```
@doV    cmpi.w  #$FE40,d5               ; V>=-448
        bge.s   @okV                    ; if greater
        move.w  #$FE40,d5               ; V= -448
        bra.s   @end                    ; save
@okV    cmpi.w  #$01CC,d5               ; V< 448
        blt.s   @end                    ; if less
        move.w  #$01C0,d5               ; V= 448

@end    move.w  d4,(a1)+                ; Save U
        move.w  d5,(a2)+                ; Save V

        cmpa.l  LS.U_ex(a6),a1
        blt.w   @do_x

        add.l   LS.Y_y(a6),a0
        add.l   LS.P_y(a6),a3

        cmpa.l  LS.U_ey(a6),a1
        blt.w   @do_y
*
        movem.l (a7)+,d4-d7/a3-a5       ; restore registers
        unlk    a6                      ; remove locals
        rts     ; return
*
        ENDFUNC
*------------------------------------
        macro
        UV16    &AU, &AV, &SP, &UV
*
        move.l  (&AU)+,&SP
        move.l  (&AV)+,&UV
        UVOVER  &SP,&UV
        lsr.l   #5,&UV
        andi.l  #$03e003e0,&SP
        andi.l  #$001F001F,&UV
        or.l    &SP,&UV                 ; UV==$00UV00UV
        swap    &UV
*
        endm

        macro
        Y16x2   &AY,&IND,&UV
*
        move.l  &AY,d2                  ; (2+) Y=Y0Y1
        lsl.l   #5,d2                   ; (4) Y=Y0XXY1XX
        andi.l  #$FC00FC00,d2           ;
        or.w    &UV,d2                  ; (2) Y=Y1UV
        move.l  (&IND,d2.w*4),d1        ; (2+) d1=0123 (Y1)
        swap    d2                      ; (4) Y=Y0XX
        or.w    &UV,d2                  ; (2) Y=Y0UV
        move.l  (&IND,d2.w*4),d2        ; (2+) d2=0123 (Y0)
*
        endm

OUT16x2 FUNC    EXPORT
*
PS      RECORD  8
table   DS.L    1
pixmap  DS.L    1
Y       DS.L    1
U       DS.L    1
V       DS.L    1
```

Engineering:KlicsCode:CompPict:Colour.a

```
width    DS.L        1
height   DS.L        1
rowByte  DS.L        1
pixmap2  DS.L        1
         ENDR
*
LS       RECORD      0,DECR
Y1       DS.L        1            ; sizeof(short)*Yrow            = 2*width
U_ex     DS.L        1            ; x end address                 = U+U_ix
U_ey     DS.L        1            ; y end address                 = U+width*height>>
U_ix     DS.L        1            ; sizeof(short)*UVrow           = width
Y_y      DS.L        1            ; sizeof(short)*Yrow            = 2*width
P_y      DS.L        1            ; 4*rowBytes-sizeof(long)*Prow  = 4*rowBytes-width
LSize    EQU         *
         ENDR
*
*        a0 - Y, a1 - U, a2 - V, a3 - pixmap, a4 - table, a5 - pixmap2
*        d0 - rgb00, d1 - rgb01, d2 - rgb10, d3 - rgb11, d4 - spare, d6 - old0, d7
*
         link        a6,#LS.LSize             ; inc, width, fend and rowend are loca
         movem.l     d4-d7/a3-a5,-(a7)        ; store registers
*
         move.l      PS.Y(a6),a0              ; Y=Yc
         move.l      PS.U(a6),a1              ; U=Uc
         move.l      PS.V(a6),a2              ; V=Vc
         move.l      PS.pixmap(a6),a3         ; pm=pixmap
         move.l      PS.table(a6),a4          ; tab=table
         adda.l      #$00020000,a4            ; tab+=32768 (longs)
         move.l      PS.pixmap2(a6),a5        ; pm2=pixmap2

         move.l      PS.width(a6),d0          ; LOAD width
         move.l      d0,LS.U_ix(a6)           ; SAVE U_ix
         move.l      PS.height(a6),d1         ; LOAD height
         mulu.w      d0,d1                    ;  width*height
         lsr.l       #1,d1                    ;  width*height/2
         add.l       a1,d1                    ;  U+width*height/2
         move.l      d1,LS.U_ey(a6)           ; SAVE U_ey
         add.l       d0,d0                    ;  width*2
         move.l      d0,LS.Y1(a6)             ; SAVE Y1
         move.l      d0,LS.Y_y(a6)            ; SAVE Y_y
         add.l       d0,d0                    ;  width*4
         move.l      PS.rowByte(a6),d1        ; LOAD rowBytes
         add.l       d1,d1                    ;  rowBytes*2
         add.l       d1,d1                    ;  rowBytes*4
         sub.l       d0,d1                    ;  rowBytes*4-width*4
         move.l      d1,LS.P_y(a6)            ; SAVE P_y

         move.l      PS.rowByte(a6),d5        ; load rowBytes
         clr.l       d6
         clr.l       d7

@do_y    move.l      LS.U_ix(a6),d0           ; LOAD U_ixB
         add.l       a1,d0                    ;  P+U_ixB
         move.l      d0,LS.U_ex(a6)           ; SAVE U_exB

@do_x    GETUV       a1,a2,d0,d4              ; d4=00UV00UV (10)

         GETY        (a0),a4,d4,d1,d2         ; calc d2,d1 pixel
         move.l      d2,(a3)+
         move.l      d1,(a3)
         add.l       d5,a3
         swap        d1
         move.l      d1,(a3)
```

```
swap        d2
move.l      d2,-(a3)
add.l       d5,a3

move.l      LS.Y1(a6),d0            ; load Yrow
GETY        (a0,d0.w),a4,d4,d1,d2   ; calc d2,d1 pixels
move.l      d2,(a3)+
move.l      d1,(a3)
add.l       d5,a3
swap        d1
move.l      d1,(a3)
swap        d2
move.l      d2,-(a3)

swap        d4                      ; next UV
addq.l      #4,a0                   ; next Ys
add.l       #12,a3

move.l      LS.Y1(a6),d0            ; load Yrow
GETY        (a0,d0.w),a4,d4,d1,d2   ; calc d2,d1 pixels
move.l      d1,(a3)
move.l      d2,-(a3)
sub.l       d5,a3
swap        d2
move.l      d2,(a3)+
swap        d1
move.l      d1,(a3)
sub.l       d5,a3

GETY        (a0)+,a4,d4,d1,d2
move.l      d1,(a3)
move.l      d2,-(a3)
swap        d2
sub.l       d5,a3
move.l      d2,(a3)+
swap        d1
move.l      d1,(a3)+

cmpa.l      LS.U_ex(a6),a1
blt.w       @do_x

add.l       LS.Y_y(a6),a0
add.l       LS.P_y(a6),a3

cmpa.l      LS.U_ey(a6),a1
blt.w       @do_y

movem.l     (a7)+,d4-d7/a3-a5       ; restore registers
unlk        a6                      ; remove locals
rts         ; return

ENDFUNC
;--------------------------------
macro
Y16         &AY,&IND,&UV

move.l      &AY,d2                  ; (2+) Y=Y0Y1
lsl.l       #5,d2                   ; (4) Y=Y0XXY1XX
andi.l      #$FC00FC00,d2           ;
or.w        &UV,d2                  ; (2) Y=Y1UV
move.l      (&IND,d2.w*4),d1        ; (2+) d1=Y1
swap        d2                      ; (4) Y=Y0XX
or.w        &UV,d2                  ; (2) Y=Y0UV
```

```
        move.l      (&IND,d2.w*4),d2          ; (2+) d2=Y0
        move.w      d1,d2                     ; (2) d2=Y0Y1
*
        endm

CUT16   FUNC    EXPORT
*
PS      RECORD      8
table   DS.L        1
pixmap  DS.L        1
Y       DS.L        1
U       DS.L        1
V       DS.L        1
width   DS.L        1
height  DS.L        1
rowByte DS.L        1
pixmap2 DS.L        1
        ENDR
*
LS      RECORD      0,DECR
Y1      DS.L        1       ; sizeof(short)*Yrow                = 2*width
U_ex    DS.L        1       ; x end address                     = U+U_ix
U_ey    DS.L        1       ; y end address                     = U+width*height>>
U_ix    DS.L        1       ; sizeof(short)*UVrow               = width
Y_y     DS.L        1       ; sizeof(short)*Yrow                = 2*width
P_y     DS.L        1       ; 2*rowBytes-sizeof(long)*Prow      = 2*rowBytes-width
LSize   EQU         *
        ENDR
*
*       a0 - Y, a1 - U, a2 - V, a3 - pixmap, a4 - table, a5 - pixmap2
*       d0 - rgb00, d1 - rgb01, d2 - rgb10, d3 - rgb11, d4 - spare, d6 - old0, d7
*
        link        a6,#LS.LSize              ; inc, width, fend and rowend are loca
        movem.l     d4-d7/a3-a5,-(a7)         ; store registers
*
        move.l      PS.Y(a6),a0               ; Y=Yc
        move.l      PS.U(a6),a1               ; U=Uc
        move.l      PS.V(a6),a2               ; V=Vc
        move.l      PS.pixmap(a6),a3          ; pm=pixmap
        move.l      PS.table(a6),a4           ; tab=table
        adda.l      #$00020000,a4             ; tab+=32768 (longs)
        move.l      PS.pixmap2(a6),a5         ; pm2=pixmap2

        move.l      PS.width(a6),d0           ; LOAD width
        move.l      d0,LS.U_ix(a6)            ; SAVE U_ix
        move.l      PS.height(a6),d1          ; LOAD height
        mulu.w      d0,d1                     ;  width*height
        lsr.l       #1,d1                     ;  width*height/2
        add.l       a1,d1                     ;  U+width*height/2
        move.l      d1,LS.U_ey(a6)            ; SAVE U_ey
        add.l       d0,d0                     ;  width*2
        move.l      d0,LS.Y1(a6)              ; SAVE Y1
        move.l      d0,LS.Y_y(a6)             ; SAVE Y_y
        move.l      PS.rowByte(a6),d1         ; LOAD rowBytes
        add.l       d1,d1                     ;  rowBytes*2
        sub.l       d0,d1                     ;  rowBytes*2-width*2
        move.l      d1,LS.P_y(a6)             ; SAVE P_y

        move.l      PS.rowByte(a6),d5         ; load rowBytes
        clr.l       d6
        clr.l       d7

@do_y   move.l      LS.U_ix(a6),d0            ; LOAD U_ixB
```

```
        add.l       a1,d0                   ;  P+U_ixB
        move.l      d0,LS.U_ex(a6)          ; SAVE U_exB

@do_x   GETUV       a1,a2,d0,d4             ; d4=00UV00UV (10)

        GETY        (a0),a4,d4,d1,d2        ; calc d2,d1 pixel
        move.w      d1,d2
        move.l      d2,(a3)
        add.l       d5,a3

        move.l      LS.Y1(a6),d0            ; load Yrow
        GETY        (a0,d0.w),a4,d4,d1,d2   ; calc d2.d1 pixels
        move.w      d1,d2
        move.l      d2,(a3)+

        swap        d4                      ; next UV
        addq.l      #4,a0                   ; next Ys

        move.l      LS.Y1(a6),d0            ; load Yrow
        GETY        (a0,d0.w),a4,d4,d1,d2   ; calc d2,d1 pixels
        move.w      d1,d2
        move.l      d2,(a3)
        sub.l       d5,a3

        GETY        (a0)+,a4,d4,d1,d2
        move.w      d1,d2
        move.l      d2,(a3)+

        cmpa.l      LS.U_ex(a6),a1
        blt.w       @do_x

        add.l       LS.Y_y(a6),a0
        add.l       LS.P_y(a6),a3

        cmpa.l      LS.U_ey(a6),a1
        blt.w       @do_y

        movem.l     (a7)+,d4-d7/a3-a5       ; restore registers
        unlk        a6                      ; remove locals
        rts                                 ; return

        ENDFUNC
;-----------------------------------
        END
```

```
*------------------------------------
*
*   © Copyright 1993 KLICS Limited
*   All rights reserved.
*
*   Written by: Adrian Lewis
*
*------------------------------------
*
*   68000 Fast RGB/YUV code
*
*------------------------------------
        include 'Traps.a'
        machine mc68030
*------------------------------------
        macro
        RGB2Y   &Apixel,&AY
*
*       d0 - pixel/r, d1 - g/2g+r, d2 - b, d3 - Y
*
        move.l  &Apixel,d0          ; pixel=*Apixel
        eor.l   #$00808080,d0       ; signed pixels

        move.b  d0,d2               ; b=pixel[3]
        ext.w   d2                  ; b is 8(16) bit

        move.w  d0,d1               ; g=pixel[2]
        asr.w   #7,d1               ; 2g is 9(16) bit

        swap    d0                  ; r=pixel[1]
        ext.w   d0                  ; r is 8(16) bit

        move.w  d2,d3               ; Y=b
        lsl.w   #3,d3               ; Y<<=3
        sub.w   d2,d3               ; Y-=b

        add.w   d0,d1               ; 2g+=r
        add.w   d1,d3               ; Y+=2g+r
        add.w   d1,d3               ; Y+=2g+r
        add.w   d1,d3               ; Y+=2g+r
        asr.w   #4,d3               ; Y>>=4
        add.w   d1,d3               ; Y+=2g+r
        move.w  d3,&AY              ; AY=Y is 10(16) bit
*
        endm
*------------------------------------
        macro
        RGB2UV  &AU,&AV
*
*       d0 - r, d2 - b, d3 - Y, d1 - U/V
*
        add.w   d0,d0               ; r is 9(16) bit
        add.w   d2,d2               ; b is 9(16) bit
        asr.w   #1,d3               ; Y is 9(16) bit
        move.w  d2,d1               ; U=b
        sub.w   d3,d1               ; U=b-Y
        move.w  d1,&AU              ; AU=U
        move.w  d0,d1               ; V=r
        sub.w   d3,d1               ; V=r-Y
        move.w  d1,&AV              ; AV=V
*
        endm
*------------------------------------
```

Engineering:KlicsCode:CompPict:Color2.a

```
        if    &TYPE('seg')≠'UNDEFINED' then
        seg         &seg
        endif

RGB2YUV2    FUNC    EXPORT
*
        link        a6,#0               ; no local variables
        movem.l     d4-d7/a3,-(a7)      ; store registers
*
        move.l      $0008(a6),a3        ; pm=pixmap
        move.l      $000C(a6),a0        ; Y=Yc
        move.l      $0010(a6),a1        ; U=Uc
        move.l      $0014(a6),a2        ; V=Vc
        move.l      $0018(a6),d7        ; fend=area
        asl.l       #2,d7               ; fend<<=2
        add.l       a3,d7               ; fend+=pm
        move.l      $001C(a6),d4        ; width_b=width
        asl.l       #2,d4               ; width_b<<=2
        move.l      $0020(a6),d5        ; inc_b=cols
        asl.l       #2,d5               ; cols<<=2
        sub.l       d4,d5               ; inc_b-=width_b
@do1    move.l      a3,d6               ; rowend=pm
        add.l       d4,d6               ; rowend+=width_b
@do2    rgb2y       (a3)+,(a0)+         ; rgb2y(pm++,Y++)
        rgb2uv      (a1)+,(a2)+         ; rgb2uv(U++,V++)
        rgb2y       (a3)+,(a0)+         ; rgb2y(pm++,Y++)
        cmpa.l      d6,a3               ; rowend>pm
        blt.s       @do2                ; while
        adda.l      d5,a3               ; pm+=inc_b
        move.l      a3,d6               ; rowend=pm
        add.l       d4,d6               ; rowend+=width_b
@do3    rgb2y       (a3)+,(a0)+         ; rgb2y(pm++,Y++)
        cmpa.l      d6,a3               ; rowend>pm
        blt.s       @do3                ; while
        adda.l      d5,a3               ; pm+=inc_b
        cmpa.l      d7,a3               ; fend>pm
        blt.w       @do1                ; while
*
        movem.l     (a7)+,d4-d7/a3      ; restore registers
        unlk        a6                  ; remove locals
        rts                             ; return
*
        ENDFUNC
*-------------------------------
        macro
        FETCHY      &AY, &Y, &R, &G, &B
*
        move.l      &AY,&Y              ; Y=*AY++
        add.l       &Y,&R               ; RR+=Y12
        add.l       &Y,&G               ; GG+=Y12
        add.l       &Y,&B               ; BB+=Y12
*
        endm
*-------------------------------
        macro
        FIXOV       &V, &SP1, &SP2
*
        move.w      &V,&SP1
        clr.b       &SP1
        andi.w      #$3FFF,&SP1
        sne         &SP1
        btst        #13,&SP1
        seq         &SP2
```

Engineering:KlicsCode:CompPict:Color2.a

```
        or.b        &SP1,&V
        and.w       &SP2,&V
        swap        &V
        move.w      &V,&SP1
        clr.b       &SP1
        andi.w      #$3FFF,&SP1
        sne         &SP1
        btst        #13,&SP1
        seq         &SP2
        or.b        &SP1,&V
        and.w       &SP2,&V
        swap        &V
*
        endm
*--------------------------------
        macro
        OVERFLOW    &A, &B, &SP1, &SP2
*
*       move.l      #$FF00FF00,&SP1         ; sp1=mask
*       move.l      &A,&SP2                 ; sp2=ovov (A)
*       and.l       &SP1,&SP2               ; sp2=o0o0 (A)
*       lsr.l       #8,&SP2                 ; sp2=0o0o (A)
*       and.l       &B,&SP1                 ; sp1=o0o0 (B)
*       or.l        &SP2,&SP1               ; sp1=oooo (BABA)
        move.l      &A,&SP1
        or.l        &B,&SP1
        andi.l      #$FF00FF00,&SP1
        beq.s       @ok                     ; if no overflow
        clr.w       &SP2                    ; AND=0
        FIXOV       &A,&SP1,&SP2            ; A1 overflow
        FIXOV       &B,&SP1,&SP2            ; B1 overflow
@ok
*
        endm
*--------------------------------
        macro
        MKRGB       &R, &G, &B, &ARGB
*
        lsl.l       #8,&G                   ; G=G0G0 (12)
        or.l        &B,&G                   ; G=GBGB (12)
        move.l      &R,&B                   ; B=0R0R (12)
        swap        &B                      ; B=0R0R (21)
        move.w      &G,&B                   ; B=0RGB (2)
        swap        &G                      ; G=GBGB (21)
        move.w      &G,&R                   ; R=0RGB (1)
        move.l      &R,&ARGB                ; *RGB++=rgb (1)
        move.l      &B,&ARGB                ; *RGB++=rgb (2)
*
        endm
*--------------------------------
        macro
        DUPVAL      &V0, &V1
*
        move.w      &V0,&V1                 ; v1=v0
        swap        &V0
        move.w      &V1,&V0                 ; dup v0
        move.l      &V0,&V1                 ; dup v1
*
        endm
*--------------------------------
        macro
        UV2RGB3     &AU,&AV
*
```

Engineering:KlicsCode:CompPict:Color2.a

```
*       d1 - ra, d2 - ga, d3 - ba, d4 - rb, d5 - gb/512, d6 - bb
*
        move.w      #512,d5                   ; d5=512
        move.w      &AU,d2                    ; U=*AU++
        add.w       d2,d2                     ; U is 10(16) bits
        move.w      d2,d3                     ; ba=U
        add.w       d3,d2                     ; ga=2U
        add.w       d3,d2                     ; ga=3U
        add.w       d5,d3                     ; ba+=512
        DUPVAL      d3,d6                     ; ba=bb=BB
        asr.w       #4,d2                     ; ga=3U>>4
        move.w      &AV,d1                    ; V=*AV++
        add.w       d1,d2                     ; ga+=V
        add.w       d1,d1                     ; ra*=2
        add.w       d5,d1                     ; ra+=512
        DUPVAL      d1,d4                     ; ra=rb=RR
        sub.w       d2,d5                     ; gb=512-ga
        DUPVAL      d5,d2                     ; ga=gb=GG
*
        endm
*--------------------------------
        if  &TYPE('seg')≠'UNDEFINED' then
        seg         &seg
        endif

YUV2RGB2    FUNC    EXPORT
*
PS      RECORD      8
pixmap  DS.L        1
Y       DS.L        1
U       DS.L        1
V       DS.L        1
area    DS.L        1
width   DS.L        1
cols    DS.L        1
        ENDR
*
LS      RECORD      0,DECR
inc     DS.L        1
width   DS.L        1
fend    DS.L        1
count   DS.L        1
LSize   EQU         *
        ENDR
*
*       a0 - Y0, a1 - Y1, a2 - U, a3 - V, a4 - pm0, a5 - pm1
*       d0..6 - used, d7 - count
*
        link        a6,#LS.LSize              ; inc, width, fend and rowend are loca
        movem.l     d4-d7/a3-a5,-(a7)         ; store registers
*
        move.l      PS.pixmap(a6),a4          ; pm0=pixmap
        move.l      a4,a5                     ; pm1=pm0
        move.l      PS.Y(a6),a0               ; Y0=Yc
        move.l      a0,a1                     ; Y1=Y0
        move.l      PS.U(a6),a2               ; U=Uc
        move.l      PS.V(a6),a3               ; V=Vc
        move.l      PS.area(a6),d7            ; fend=area
        lsl.l       #2,d7                     ; fend<<=2
        add.l       a4,d7                     ; fend+=pm0
        move.l      d7,LS.fend(a6)            ; save fend
        move.l      PS.width(a6),d5           ; width=width
        move.l      d5,d7                     ; count=width
```

Engineering:KlicsCode:CompPict:Color2.a

```
        asr.l    #1,d7                    ; count>>=1
        subq.l   #1,d7                    ; count-=1
        move.l   d7,PS.width(a6)          ; save width
        add.l    d5,d5                    ; width*=2
        add.l    d5,a1                    ; Y1+=width
        add.l    d5,d5                    ; width*=2
        move.l   d5,LS.width(a6)          ; save width
        move.l   PS.cols(a6),d4           ; inc=cols
        lsl.l    #2,d4                    ; inc<<=2
        add.l    d4,a5                    ; pm1+=inc
        add.l    d4,d4                    ; cols*=2
        sub.l    d5,d4                    ; inc now 2*cols-width bytes
        move.l   d4,LS.inc(a6)            ; save inc
@do     UV2RGB3  (a2)+,(a3)+              ; uv2rgb(*U++,*V++)
        FETCHY   (a0)+,d0,d1,d2,d3        ; add Ya to RGB values
        FETCHY   (a1)+,d0,d4,d5,d6        ; add Yb to RGB values
        move.w   #$3FFF,d0                ; d0=mask
        lsr.l    #2,d1                    ; d1 8(16) bits
        and.w    d0,d1                    ; d1 masked
        lsr.l    #2,d2                    ; d2 8(16) bits
        and.w    d0,d2                    ; d2 masked
        lsr.l    #2,d3                    ; d3 8(16) bits
        and.w    d0,d3                    ; d3 masked
        lsr.l    #2,d4                    ; d4 8(16) bits
        and.w    d0,d4                    ; d4 masked
        lsr.l    #2,d5                    ; d5 8(16) bits
        and.w    d0,d5                    ; d5 masked
        lsr.l    #2,d6                    ; d6 8(16) bits
        and.w    d0,d6                    ; d6 masked
        move.l   d1,d0
        or.l     d2,d0
        or.l     d3,d0
        or.l     d4,d0
        or.l     d5,d0
        or.l     d6,d0
        andi.l   #$FF00FF00,d0
        bne.s    @over                    ; if overflow
@ok     MKRGB    d1,d2,d3,(a4)+           ; save RGBa
        MKRGB    d4,d5,d6,(a5)+           ; save RGBb
        dbf      d7,@do                   ; while
        adda.l   LS.inc(a6),a4            ; pm0+=inc
        adda.l   LS.inc(a6),a5            ; pm1+=inc
        adda.l   LS.width(a6),a0          ; Y0+=width
        exg.l    a0,a1                    ; Y1<->Y0
        move.l   PS.width(a6),d7          ; count=width
        cmpa.l   LS.fend(a6),a4           ; pm0<fend
        blt.w    @do                      ; while

        movem.l  (a7)+,d4-d7/a3-a5        ; restore registers
        unlk     a6                       ; remove locals
        rts                               ; return
@over   move.l   d7,LS.count(a6)          ; save count
        clr.w    d7                       ; AND=0

        FIXOV    d1,d0,d7                 ; A overflow
        FIXOV    d2,d0,d7                 ; B overflow
        FIXOV    d3,d0,d7                 ; A overflow
        FIXOV    d4,d0,d7                 ; B overflow
        FIXOV    d5,d0,d7                 ; A overflow
        FIXOV    d6,d0,d7                 ; B overflow
        move.l   LS.count(a6),d7          ; restore count
        bra      @ok
```

```
        ENDFUNC
*--------------------------------
        if  &TYPE('seg')≠'UNDEFINED' then
        seg         &seg
        endif

GREY2Y  FUNC    EXPORT
*
PS      RECORD      8
pixmap  DS.L        1
Y       DS.L        1
area    DS.L        1
width   DS.L        1
cols    DS.L        1
        ENDR
*
*   d0 - vvvv, d1 - v0v1, d2 - v2v3, d3 - xor, d4 - width, d5 - inc, d6 - rowend,
*   a0 - pm, a1 - Y
*
        link        a6,#0                   ; no local variables
        movem.l     d4-d7,-(a7)             ; store registers
*
        move.l      PS.pixmap(a6),a0        ; pm=pixmap
        move.l      PS.Y(a6),a1             ; Y=Yc
        move.l      PS.area(a6),d7          ; fend=area
        add.l       a0,d7                   ; fend+=pm
        move.l      PS.width(a6),d4         ; width_b=width
        move.l      PS.cols(a6),d5          ; inc_b=cols
        sub.l       d4,d5                   ; inc_b-=width_b
        move.l      #$7F7F7F7F,d3           ; xor=$7F7F7F7F
@do1    move.l      a0,d6                   ; rowend=pm
        add.l       d4,d6                   ; rowend+=width_b
@do2    move.l      (a0)+,d0                ; vvvv=*pm
        eor.l       d3,d0                   ; vvvv is signed
        move.w      d0,d2                   ; d2=v2v3
        asr.w       #6,d2                   ; d2=v2 (10 bits)
        swap        d2                      ; d2=v2??
        move.b      d0,d2                   ; d2=v2v3
        ext.w       d2                      ; v3 extended
        lsl.w       #2,d2                   ; d2=v2v3 (10 bits)
        swap        d0                      ; d0=v0v1
        move.w      d0,d1                   ; d1=v0v1
        asr.w       #6,d1                   ; d1=v0 (10 bits)
        swap        d1                      ; d1=v0??
        move.b      d0,d1                   ; d1=v0v1
        ext.w       d1                      ; v1 extended
        lsl.w       #2,d1                   ; d1=v0v1 (10 bits)
        move.l      d1,(a1)+                ; *Y=d1
        move.l      d2,(a1)+                ; *Y=d2
        cmpa.l      d6,a0                   ; rowend>pm
        blt.s       @do2                    ; while
        adda.l      d5,a0                   ; pm+=inc_b
        cmpa.l      d7,a0                   ; fend>pm
        blt.s       @do1                    ; while
*
        movem.l     (a7)+,d4-d7             ; restore registers
        unlk        a6                      ; remove locals
        rts                                 ; return
*
        ENDFUNC
*--------------------------------
        if  &TYPE('seg')≠'UNDEFINED' then
        seg         &seg
```

Engineering:KlicsCode:CompPict:Color2.a

```
        endif

Y2GREY  FUNC    EXPORT
*
PS      RECORD      8
pixmap  DS.L        1
Y       DS.L        1
height  DS.L        1
width   DS.L        1
cols    DS.L        1
        ENDR
*
*   d0- spare, d1 - v43, d2 - v21, d3 - spare, d4 - width, d5 - inc, d6 - count, d
*   a0 - pm, a1 - Y
*
        link        a6,#0                   ; no local variables
        movem.l     d4-d7,-(a7)             ; store registers
*
        move.l      PS.pixmap(a6),a0        ; pm=pixmap
        move.l      PS.Y(a6),a1             ; Y=Yc
        move.l      PS.height(a6),d7        ; long height
        subq.l      #1,d7                   ; height-=1
        move.l      PS.width(a6),d4         ; long width
        move.l      PS.cols(a6),d5          ; long inc=cols
        sub.l       d4,d5                   ; inc-=width
        lsr.l       #2,d4                   ; width>>=2 (read 4 values)
        subq.l      #1,d4                   ; width-=1
@do1    move.l      d4,d6                   ; count=width
@do2    move.l      (a1)+,d0                ; d0=x4x3
        move.l      (a1)+,d1                ; d1=x2x1
        move.l      #$01FF01FF,d2           ; d2=511
        move.l      d2,d3                   ; d3=511
        sub.l       d0,d2                   ; unsigned d2
        sub.l       d1,d3                   ; unsigned d3
        lsr.l       #2,d2
        lsr.l       #2,d3
        move.l      d2,d0
        or.l        d3,d0
        andi.l      #$3F003F00,d0
        bne.s       @over                   ; if no overflow
@ok     lsl.w       #8,d3                   ; d3=0210
        lsl.w       #8,d2                   ; d2=0430
        lsr.l       #8,d3                   ; d3=0021
        lsl.l       #8,d2                   ; d2=4300
        or.l        d3,d2                   ; d2=4321
        move.l      d2,(a0)+                ; *pm=d2
        dbf         d6,@do2                 ; while -1!=--count
        adda.l      d5,a0                   ; pm+=inc_b
        dbf         d7,@do1                 ; while -1!=--height
*
        movem.l     (a7)+,d4-d7             ; restore registers
        unlk        a6                      ; remove locals
        rts                                 ; return
@over   clr.w       d1                      ; AND=0
        FIXOV       d2,d0,d1                ; A overflow
        FIXOV       d3,d0,d1                ; B overflow
        bra.s       @ok
*
        ENDFUNC
*-----------------------------------
        macro
        GGG         &V,&SP1,&SP2,&AV
*
```

Engineering:KlicsCode:CompPict:Color2.a

```
        move.l      &V,&SP2                     ; SP2=0102
        lsl.l       #8,&SP2                     ; SP2=1020
        or.l        &V,&SP2                     ; SP2=1122
        move.l      &V,&SP1                     ; SP1=0102
        swap        &SP1                        ; SP1=0201
        move.w      &SP2,&SP1                   ; SP1=0222
        swap        &SP2                        ; SP2=2211
        move.w      &SP2,&V                     ; V=0111
        move.l      &V,&AV                      ; *pm=V
        move.l      &SP1,&AV                    ; *pm=SP1
*
        endm
*---------------------------------
        if   &TYPE('seg')≠'UNDEFINED' then
        seg         &seg
        endif

Y2GGG   FUNC    EXPORT
*
PS      RECORD      8
pixmap  DS.L        1
Y       DS.L        1
lines   DS.L        1
width   DS.L        1
cols    DS.L        1
        ENDR
*
*   d0 - v, d4 - width, d5 - inc, d6 - count, d7 - lines
*   a0 - pm, a1 - Y
*
        link        a6,#0                       ; no local variables
        movem.l     d4-d7,-(a7)                 ; store registers
*
        move.l      PS.pixmap(a6),a0            ; pm=pixmap
        move.l      PS.Y(a6),a1                 ; Y=Yc
        move.l      PS.lines(a6),d7             ; long lines
        subq.l      #1,d7                       ; lines-=1
        move.l      PS.width(a6),d4             ; long width
        move.l      PS.cols(a6),d5              ; inc=cols
        sub.l       d4,d5                       ; inc-=width
        lsl.l       #2,d5                       ; inc (bytes)
        lsr.l       #2,d4                       ; width>>=2
        subq.l      #1,d4                       ; width-=1
@do1    move.l      d4,d6                       ; count=width
@do2    move.l      (a1)+,d0                    ; d0=x1x2 (10 bits signed)
        move.l      (a1)+,d1                    ; d1=x3x4 (10 bits)
        move.l      #$02000200,d3               ; d3=plus
        add.l       d3,d0                       ; d0=x1x2 (unsigned)
        add.l       d3,d1                       ; d1=x3x4 (unsigned)
        lsr.l       #2,d0                       ; d0=x1x2 (10,8 bits)
        lsr.l       #2,d1                       ; d1=x3x4 (10,8 bits)
        move.w      #$3FFF,d2                   ; d2=mask
        and.w       d2,d0                       ; mask d0
        and.w       d2,d1                       ; mask d1
        move.l      d0,d2
        or.l        d1,d2
        andi.l      #$FF00FF00,d2
        bne.s       @over                       ; if no overflow
@ok     GGG         d0,d2,d3,(a0)+
        GGG         d1,d2,d3,(a0)+
        dbf         d6,@do2                     ; while -1!=--count
        adda.l      d5,a0                       ; pm+=inc_b
        dbf         d7,@do1                     ; while -1!=--lines
```

➡ Engineering:KlicsCode:CompPict:Color2.a

```
;
        movem.l     (a7)+,d4-d7            ; restore registers
        unlk        a6                     ; remove locals
        rts                                ; return
@over   clr.w       d3                     ; AND=0
        FIXOV       d0,d2,d3               ; A overflow
        FIXOV       d1,d2,d3               ; B overflow
        bra.w       @ok
;
        ENDFUNC
;-------------------------------
        macro
        MKRGB2      &R, &G, &B, &ARGB, &ROW, &XX
;
        lsl.l       #8,&G                  ; G=G0G0 (12)
        or.l        &B,&G                  ; G=GBGB (12)
        move.l      &R,&B                  ; B=0R0R (12)
        swap        &B                     ; B=0R0R (21)
        move.w      &G,&B                  ; B=0RGB (2)
        swap        &G                     ; G=GBGB (21)
        move.w      &G,&R                  ; R=0RGB (1)

        andi.l      #$FFFEFEFE,&R          ; 7 bits for interpolation
        andi.l      #$FFFEFEFE,&B          ; 7 bits for interpolation

        move.l      &R,&G                  ; G=RGB(1)
        add.l       &B,&G                  ; G+=RGB(2)
        lsr.l       #1,&G                  ; G/=2

        move.l      &B,&XX                 ; XX=RGB(2)
        sub.l       &R,&XX                 ; XX-=RGB(1)
        lsr.l       #1,&XX                 ; XX/=2
        add.l       &B,&XX                 ; XX+=B

        move.l      &R,(&ARGB)+            ; *RGB++=rgb (1)
        move.l      &G,(&ARGB)+            ; *RGB++=rgb (1.5)
        move.l      &B,(&ARGB)+            ; *RGB++=rgb (2)
        move.l      &B,(&ARGB)+            ; *RGB++=rgb (2.5)

        add.l       &ROW,&ARGB
        sub.l       #16,&ARGB

        move.l      &R,(&ARGB)+            ; *RGB++=rgb (1)
        move.l      &G,(&ARGB)+            ; *RGB++=rgb (1.5)
        move.l      &B,(&ARGB)-            ; *RGB++=rgb (2)
        move.l      &B,(&ARGB)+            ; *RGB++=rgb (2.5)

        sub.l       &ROW,&ARGB
;
        endm
;-------------------------------
        if  &TYPE('seg')#'UNDEFINED' then
        seg         &seg
        endif

YUV2RGB3    FUNC    EXPORT
;
PS      RECORD      8
pixmap  DS.L        1
Y       DS.L        1
U       DS.L        1
V       DS.L        1
area    DS.L        1
```

Engineering:KlicsCode:CompPict:Color2.a

```
width   DS.L        1
cols    DS.L        1
        ENDR
*
LS      RECORD      0,DECR
inc     DS.L        1
width   DS.L        1
fend    DS.L        1
count   DS.L        1
row     DS.L        1
LSize   EQU         *
        ENDR
*
*       a0 - Y0, a1 - Y1, a2 - U, a3 - V, a4 - pm0, a5 - pm1
*       d0..6 - used, d7 - count
*
        link        a6,#LS.LSize            ; inc, width, fend and rowend are loca
        movem.l     d4-d7/a3-a5,-(a7)       ; store registers
*
        move.l      PS.pixmap(a6),a4        ; pm0=pixmap
        move.l      a4,a5                   ; pm1=pm0
        move.l      PS.Y(a6),a0             ; Y0=Yc
        move.l      a0,a1                   ; Y1=Y0
        move.l      PS.U(a6),a2             ; U=Uc
        move.l      PS.V(a6),a3             ; V=Vc
        move.l      PS.area(a6),d7          ; fend=area
        lsl.l       #2,d7                   ; fend<<=2
        add.l       a4,d7                   ; fend+=pm0
        move.l      d7,LS.fend(a6)          ; save fend
        move.l      PS.width(a6),d5         ; width=width
        move.l      d5,d7                   ; count=width
        asr.l       #1,d7                   ; count>>=1
        subq.l      #1,d7                   ; count-=1
        move.l      d7,PS.width(a6)         ; save width
        add.l       d5,d5                   ; width*=2
        add.l       d5,a1                   ; Y1+=width
        add.l       d5,d5                   ; width*=2
        move.l      d5,LS.width(a6)         ; save width
        move.l      PS.cols(a6),d4          ; inc=cols
        lsl.l       #2,d4                   ; inc<<=2
        move.l      d4,LS.row(a6)           ;*NEW save row
        add.l       d4,a5                   ; pm1+=inc
        add.l       d4,a5                   ;*NEW pm1+=inc
        add.l       d4,d4                   ; cols*=2
        add.l       d4,d4                   ;*NEW cols*=2
        sub.l       d5,d4                   ; inc now 4*cols-width bytes
        sub.l       d5,d4                   ;*NEW  inc now 4*cols-width bytes (wid
        move.l      d4,LS.inc(a6)           ; save inc
@do     UV2RGB3     (a2)+,(a3)+             ; uv2rgb(*U++,*V++)

        FETCHY      (a0)+,d0,d1,d2,d3       ; add Ya to RGB values
        FETCHY      (a1)+,d0,d4,d5,d6       ; add Yb to RGB values

        move.w      #$3FFF,d0               ; d0=mask
        lsr.l       #2,d1                   ; d1 8(16) bits
        and.w       d0,d1                   ; d1 masked
        lsr.l       #2,d2                   ; d2 8(16) bits
        and.w       d0,d2                   ; d2 masked
        lsr.l       #2,d3                   ; d3 8(16) bits
        and.w       d0,d3                   ; d3 masked
        lsr.l       #2,d4                   ; d4 8(16) bits
        and.w       d0,d4                   ; d4 masked
        lsr.l       #2,d5                   ; d5 8(16) bits
```

Engineering:KlicsCode:CompPict:Color2.a

```
        and.w       d0,d5                        ; d5 masked
        lsr.l       #2,d6                        ; d6 8(16) bits
        and.w       d0,d6                        ; d6 masked

        move.l      d1,d0
        or.l        d2,d0
        or.l        d3,d0
        or.l        d4,d0
        or.l        d5,d0
        or.l        d6,d0
        andi.l      #$FF00FF00,d0
        bne.w       @over                        ; if overflow

@ok     MKRGB2      d1,d2,d3,a4,LS.row(a6),d0    ;*NEW save RGBa
        MKRGB2      d4,d5,d6,a5,LS.row(a6),d0    ;*NEW save RGBb
        dbf         d7,@do                       ; while
        adda.l      LS.inc(a6),a4                ; pm0+=inc
        adda.l      LS.inc(a6),a5                ; pm1+=inc
        adda.l      LS.width(a6),a0              ; Y0+=width
        exg.l       a0,a1                        ; Y1<->Y0
        move.l      PS.width(a6),d7              ; count=width
        cmpa.l      LS.fend(a6),a4               ; pm0<fend
        blt.w       @do                          ; while

        movem.l     (a7)+,d4-d7/a3-a5            ; restore registers
        unlk        a6                           ; remove locals
        rts                                      ; return
@over   move.l      d7,LS.count(a6)              ; save count
        clr.w       d7                           ; AND=0
        FIXOV       d1,d0,d7                     ; A overflow
        FIXOV       d2,d0,d7                     ; B overflow
        FIXOV       d3,d0,d7                     ; A overflow
        FIXOV       d4,d0,d7                     ; B overflow
        FIXOV       d5,d0,d7                     ; A overflow
        FIXOV       d6,d0,d7                     ; B overflow
        move.l      LS.count(a6),d7              ; restore count
        bra         @ok

        ENDFUNC
;------------------------------------
        macro
        FETCHY2     &AY, &Y, &R, &G, &B

        move.l      &AY,&Y                       ;Y
        asr.w       #2,&Y
        swap        &Y
        asr.w       #2,&Y                        ;Y is      -128 to +127
        swap        &Y                           ;RED, Get (Y+ 2V + 512) for Red = (Y +
        add.l       &Y,&R                        ;GREEN, Get (Y + (512 - (6U/16)) - V)
        add.l       &Y,&G                        ;BLUE,Get (Y + (2U + 512) for Blue = (
        add.l       &Y,&B

        endm
;------------------------------------
        macro
        UV2RGB4     &AU,&AV

        move.w      &AU,d2                       ; U
        and.w       #$03FF,d2
        move.l      (a6,d2.w*8),d3               ;BLUE,Get (2U + 512)/4 for Blue = (Y +
        move.l      d3,d6                        ;Dup for second pair
        move.l      4(a6,d2.w*8),d5              ;GREEN, Get (512 - (6U/16))/4 for Gree
        move.w      &AV,d1                       ; V
```

Engineering:KlicsCode:CompPict:Color2.a

```
        move.w      d1,d4
        asr.w       #2,d1
        sub.w       d1,d5                   ;GREEN, Get (512 - (6U/16) - V)/4 for
        move.w      d5,d2
        swap        d5
        move.w      d2,d5
        move.l      d5,d2                   ;Dup for second pair

        and.w       #$03FF,d4
        move.l      (a6,d4.w*8),d4          ;RED, Get (2V + 512)/4 for Red = (Y +
        move.l      d4,d1
*
        endm
*-----------------------------------

MKRGB2SUB   FUNC    EXPORT
*
        MKRGB2      d1,d2,d3,a4,d7,d0   ;*NEW save RGBa
        MKRGB2      d4,d5,d6,a5,d7,d0   ;*NEW save RGBb
        rts
*
        ENDFUNC

OVERSUB FUNC        EXPORT
*
        move.l      d1,d0
        or.l        d2,d0
        or.l        d3,d0
        or.l        d4,d0
        or.l        d5,d0
        or.l        d6,d0
        andi.l      #$FF00FF00,d0
        bne.s       @over                   ; if overflow
@ok     rts
@over   move.l      d7,-(sp)                ; save count
        clr.w       d7                      ; AND=0
        FIXOV       d1,d0,d7                ; A overflow
        FIXOV       d2,d0,d7                ; B overflow
        FIXOV       d3,d0,d7                ; A overflow
        FIXOV       d4,d0,d7                ; B overflow
        FIXOV       d5,d0,d7                ; A overflow
        FIXOV       d6,d0,d7                ; B overflow
        move.l      (sp)+,d7                ; restore count
        bra         @ok
*
        ENDFUNC

UV2RGB4SUB  FUNC    EXPORT
*
        UV2RGB4     (a2)+,(a3)+             ; uv2rgb(*U++,*V++)
        rts
*
        ENDFUNC

FETCHY2SUB  FUNC    EXPORT
*
        FETCHY2     (a0)+,d0,d1,d2,d3       ; add Ya to RGB values
        FETCHY2     (a1)+,d0,d4,d5,d6       ; add Yb to RGB values
        rts
*
        ENDFUNC

        if  &TYPE('seg')#'UNDEFINED' then
```

Engineering:KlicsCode:CompPict:Color2.a

```
        seg             &seg
        endif

YUV2RGB5    FUNC        EXPORT
*
PS      RECORD          8
Table   DS.L            1
pixmap  DS.L            1
Y       DS.L            1
U       DS.L            1
V       DS.L            1
area    DS.L            1
width   DS.L            1
cols    DS.L            1
        ENDR
*
LS      RECORD          0,DECR
inc     DS.L            1
width   DS.L            1
fend    DS.L            1
count   DS.L            1
row     DS.L            1
LSize   EQU             *
        ENDR
*
*       a0 - Y0, a1 - Y1, a2 - U, a3 - V, a4 - pm0, a5 - pm1
*       d0..6 - used, d7 - count
*
        link            a6,#LS.LSize            ; inc, width, fend and rowend are loca
        movem.l         d4-d7/a3-a5,-(a7)       ; store registers
*
        move.l          PS.pixmap(a6),a4        ; pm0=pixmap
        move.l          a4,a5                   ; pm1=pm0
        move.l          PS.Y(a6),a0             ; Y0=Yc
        move.l          a0,a1                   ; Y1=Y0
        move.l          PS.U(a6),a2             ; U=Uc
        move.l          PS.V(a6),a3             ; V=Vc
        move.l          PS.area(a6),d7          ; fend=area
        lsl.l           #2,d7                   ; fend<<=2
        add.l           a4,d7                   ; fend+=pm0
        move.l          d7,LS.fend(a6)          ; save fend
        move.l          PS.width(a6),d5         ; width=width
        move.l          d5,d7                   ; count=width
        asr.l           #1,d7                   ; count>>=1
        subq.l          #1,d7                   ; count-=1
        move.l          d7,PS.width(a6)         ; save width

        add.l           d5,d5                   ; width*=2
        add.l           d5,a1                   ; Y1+=width
        add.l           d5,d5                   ; width*=2
        move.l          d5,LS.width(a6)         ; save width
        move.l          PS.cols(a6),d4          ; inc=cols
        lsl.l           #2,d4                   ; inc<<=2
        move.l          d4,LS.row(a6)           ;*NEW save row
        add.l           d4,a5                   ; pm1+=inc
        add.l           d4,a5                   ;*NEW pm1+=inc
        add.l           d4,d4                   ; cols*=2
        add.l           d4,d4                   ;*NEW cols*=2
        sub.l           d5,d4                   ; inc now 4*cols-width bytes
        sub.l           d5,d4                   ;*NEW  inc now 4*cols-width bytes (wid
        move.l          d4,LS.inc(a6)           ; save inc

@do     move.l          d7,-(sp)
```

Engineering:KlicsCode:CompPict:Color2.a

```
        move.l      a6,-(sp)
        move.l      LS.row(a6),d7
        move.l      PS.Table(a6),a6
        UV2RGB4     (a2)+,(a3)+             ; uv2rgb(*U++,*V++)

        FETCHY2     (a0)+,d0,d1,d2,d3       ; add Ya to RGB values
        FETCHY2     (a1)+,d0,d4,d5,d6       ; add Yb to RGB values

        move.l      d1,d0
        or.l        d2,d0
        or.l        d3,d0
        or.l        d4,d0
        or.l        d5,d0
        or.l        d6,d0
        andi.l      #$FF00FF00,d0
        bne.w       @over                   ; if overflow

@ok     MKRGB2      d1,d2,d3,a4,d7,d0       ;*NEW save RGBa
        MKRGB2      d4,d5,d6,a5,d7,d0       ;*NEW save RGBb
        move.l      (sp)+,a6
        move.l      (sp)+,d7

        dbf         d7,@do                  ; while

        adda.l      LS.inc(a6),a4           ; pm0+=inc
        adda.l      LS.inc(a6),a5           ; pm1+=inc
        adda.l      LS.width(a6),a0         ; Y0+=width
        exg.l       a0,a1                   ; Y1<->Y0
        move.l      PS.width(a6),d7         ; count=width
        cmpa.l      LS.fend(a6),a4          ; pm0<fend
        blt.s       @do                     ; while
*
        movem.l     (a7)+,d4-d7/a3-a5       ; restore registers
        unlk        a6                      ; remove locals
        rts                                 ; return
@over   move.l      d7,LS.count(a6)         ; save count
        clr.w       d7                      ; AND=0
        FIXOV       d1,d0,d7                ; A overflow
        FIXOV       d2,d0,d7                ; B overflow
        FIXOV       d3,d0,d7                ; A overflow
        FIXOV       d4,d0,d7                ; B overflow
        FIXOV       d5,d0,d7                ; A overflow
        FIXOV       d6,d0,d7                ; B overflow
        move.l      LS.count(a6),d7         ; restore count
        bra         @ok
*
        ENDFUNC
*------------------------------------
        END
```

Engineering:KlicsCode:CompPict:Clut.c

```
/*****************************
 *
 * © Copyright 1993 KLICS Limited
 * All rights reserved.
 *
 * Written by: Adrian Lewis
 *
 *****************************/
/*
    Analyse CLUT setup and pick appropriate
    YUV->RGB converter/display driver. Create
    any tables necessary.
*/

#include <QuickDraw.h>
#include <Memory.h>

#define Y_LEVELS    64
#define UV_LEVELS   16

#define absv(v) ((v)<0?-(v):(v))
#define NewPointer(ptr,type,size) \
    saveZone=GetZone(); \
    SetZone(SystemZone()); \
    if (nil==(ptr=(type)NewPtr(size))) { \
        SetZone(ApplicZone()); \
        if (nil==(ptr=(type)NewPtr(size))) { \
            SetZone(saveZone); \
            return(MemoryError()); \
        } \
    } \
    SetZone(saveZone);

typedef struct {
    char    y, u, v;
} YUV_Clut;

/*
unsigned char *
ColourClut(CTabHandle clut)
{
    int     size, y, u, v, r, g, b, i;
    unsigned char *table;
    YUV_Clut    *yuv_clut;

    size=(*clut)->ctSize;
    table=(unsigned char *)NewPtr(Y_LEVELS*UV_LEVELS*UV_LEVELS);
    yuv_clut=(YUV_Clut *)NewPtr(size*sizeof(YUV_Clut));

    for(i=0;i<=size;i++) {
        r=((*clut)->ctTable[i].rgb.red>>8)-128;
        g=((*clut)->ctTable[i].rgb.green>>8)-128;
        b=((*clut)->ctTable[i].rgb.blue>>8)-128;

        yuv_clut[i].y= (306*r + 601*g + 117*b)>>10;
        yuv_clut[i].u= (512*r - 429*g - 83*b)>>10;
        yuv_clut[i].v= (-173*r - 339*g + 512*b)>>10;
    }
    for(y=-Y_LEVELS/2;y<Y_LEVELS/2-1;y++)
    for(u=-UV_LEVELS/2;u<UV_LEVELS/2-1;u++)
    for(v=-UV_LEVELS/2;v<UV_LEVELS/2-1;v++) {
        int     index,error,error2,points, Y, U, V;
```

Engineering:KlicsCode:CompPict:Clut.c

```
        Y=y<<4;
        U=u<<5;
        V=v<<5;

        index=0;
        error=131072;
        error2=131072;
        points=0;
        for(i=0;i<=size;i++) {
            int pts=0, err=0;

            if (yuv_clut[i].y>=Y && yuv_clut[i].y<Y+16)
                pts+=1;
            err+=absv(yuv_clut[i].y-Y);

            if (yuv_clut[i].u>=U && yuv_clut[i].u<U+32)
                pts+=1;
            err+=absv(yuv_clut[i].u-U);

            if (yuv_clut[i].v>=V && yuv_clut[i].v<V+32)
                pts+=1;
            err+=absv(yuv_clut[i].v-V);

            if (pts>points || (pts==points && err<error)) {
                error=err;
                index=i;
                points=pts;
            }
        }
        i=((y&0x1F)<<8)|((u&0xF)<<4)|(v&0xF);
        table[i]=(unsigned char)index;
    }
    DisposePtr((Ptr)yuv_clut);
    return table;
}*/

typedef union {
    long    pixel;
    unsigned char   rgb[4];
} Pixel;
/*
unsigned long *
ColourClut(CTabHandle clut)
{
    long    size, y, u, v, r, g, b, ro, go, bo,i;
    Pixel   *table;

    size=(*clut)->ctSize;
    table=(Pixel *)NewPtr(Y_LEVELS*UV_LEVELS*UV_LEVELS*sizeof(long));

    for(y=-Y_LEVELS/2;y<Y_LEVELS/2-1;y++)
    for(u=-UV_LEVELS/2;u<UV_LEVELS/2-1;u++)
    for(v=-UV_LEVELS/2;v<UV_LEVELS/2-1;v++) {
        Pixel   px;
        long    base, dith;

        r = 32768L + ((y<<9) + 1436L*u <<2);
        g = 32768L + ((y<<9) - 731L*u - 352L*v <<2);
        b = 32768L + ((y<<9) + 1815L*v <<2);

        r=r<0?0:r>65534?65534:r;
        g=g<0?0:g>65534?65534:g;
        b=b<0?0:b>65534?65534:b;
```

```
        ro=r%13107; r=r/13107;
        go=g%13107; g=g/13107;
        bo=b%13107; b=b/13107;

        base=215-(36*r+6*g+b);

        dith=base-(ro>2621?36:0)-(go>7863?6:0)-(bo>10484?1:0);
        px.rgb[0]=dith==215?255:dith;

        dith=base-(ro>5242?36:0)-(go>10484?6:0)-(bo>2621?1:0);
        px.rgb[1]=dith==215?255:dith;

        dith=base-(ro>7863?36:0)-(go>2621?6:0)-(bo>5242?1:0);
        px.rgb[2]=dith==215?255:dith;

        dith=base-(ro>10484?36:0)-(go>5242?6:0)-(bo>7863?1:0);
        px.rgb[3]=dith==215?255:dith;

        i=((y&0x3F)<<8)|((u&0xF)<<4)|(v&0xF);

        table[i].pixel=px.pixel;
    }
    return (unsigned long*)table;
}*/

typedef struct {
    long    red, green, blue;
} RGBError;

OSErr ColourClut(Pixel **table)
{
    long    y, u, v, r, g, b, i;
    RGBError    *err;
    THz     saveZone;

    NewPointer(*table,Pixel*,Y_LEVELS*UV_LEVELS*UV_LEVELS*sizeof(long)); /* 64k ta
    NewPointer(err,RGBError*,Y_LEVELS*UV_LEVELS*UV_LEVELS*sizeof(RGBError));

    for(i=0;i<4;i++)
    for(y=-Y_LEVELS/2;y<Y_LEVELS/2;y++)
    for(u=-UV_LEVELS/2;u<UV_LEVELS/2;u++)
    for(v=-UV_LEVELS/2;v<UV_LEVELS/2;v++) {
        RGBColor    src, dst;
        long    index,in;

        index=((y&0x3F)<<8)|((u&0xF)<<4)|(v&0xF);

        r = 32768L + ((y<<9) + (1436L*u) <<2);
        g = 32768L + ((y<<9) - (731L*u) - (352L*v) <<2);
        b = 32768L + ((y<<9) + (1815L*v) <<2);

        if (i>0) {
            r-=err[index].red;
            g-=err[index].green;
            b-=err[index].blue;
        }

        src.red=r<0?0:r>65534?65534:r;
        src.green=g<0?0:g>65534?65534:g;
        src.blue=b<0?0:b>65534?65534:b;

        (*table)[index].rgb[i]=(unsigned char)Color2Index(&src);
```

➛ Engineering:KlicsCode:CompPict:Clut.c

```
        Index2Color((*table)[index].rgb[i],&dst);
        err[index].red=dst.red-src.red;
        err[index].green=dst.green-src.green;
        err[index].blue=dst.blue-src.blue;
    }
    DisposePtr((Ptr)err);
    return(noErr);
}

typedef struct {
    short   pel[2];
} Pix16;

typedef struct {
    unsigned char   pel[4];
} Pix8;

#define YS  64
#define UVS 32

OSErr Colour8(Pix8 **table)
{
    long    y, u, v, r, g, b, i;
    RGBError    *err;
    THz     saveZone;

    NewPointer(*table,Pix8*,YS*UVS*UVS*sizeof(Pix8)); /* 128k table */
    NewPointer(err,RGBError*,YS*UVS*UVS*sizeof(RGBError));

    for(i=0;i<4;i++)
    for(y=-YS/2;y<YS/2;y++)
    for(u=-UVS/2;u<UVS/2;u++)
    for(v=-UVS/2;v<UVS/2;v++) {
        RGBColor    src, dst;
        long    index;

        index=(y<<10)|((u&0x1F)<<5)|(v&0x1F);

        r = 32768L + ((y<<10) + (1436L*u) <<1);
        g = 32768L + ((y<<10) - (731L*u) - (352L*v) <<1);
        b = 32768L + ((y<<10) + (1815L*v) <<1);

        if (i>0) {
            r-=err[32768+index].red;
            g-=err[32768+index].green;
            b-=err[32768+index].blue;
        }

        src.red=r<0?0:r>65534?65534:r;
        src.green=g<0?0:g>65534?65534:g;
        src.blue=b<0?0:b>65534?65534:b;

        (*table)[32768+index].pel[i]=(unsigned char)Color2Index(&src);
        Index2Color((*table)[32768+index].pel[i],&dst);

        err[32768+index].red=dst.red-src.red;
        err[32768+index].green=dst.green-src.green;
        err[32768+index].blue=dst.blue-src.blue;
    }
    DisposePtr((Ptr)err);
    return(noErr);
}
```

Engineering:KlicsCode:CompPict:Clut.c

```
OSErr Colour16(Pix16 **table)
{
    long    y, u, v, r, g, b, i;
    RGBError    *err;
    THz     saveZone;

    NewPointer(*table,Pix16*,YS*UVS*UVS*sizeof(Pix16)); /* 128k table */
    NewPointer(err,RGBError*,YS*UVS*UVS*sizeof(RGBError));

    for(i=0;i<2;i++)
    for(y=-YS/2;y<YS/2;y++)
    for(u=-UVS/2;u<UVS/2;u++)
    for(v=-UVS/2;v<UVS/2;v++) {
        RGBColor    src, dst;
        long    index;

        index=(y<<10)|((u&0x1F)<<5)|(v&0x1F);

        r = 32768L + ((y<<10) + (1436L*u) <<1);
        g = 32768L + ((y<<10) - (731L*u) - (352L*v) <<1);
        b = 32768L + ((y<<10) + (1815L*v) <<1);

        if (i>0) {
            r-=err[32768+index].red;
            g-=err[32768+index].green;
            b-=err[32768+index].blue;
        }

        src.red=r<0?0:r>65534?65534:r;
        src.green=g<0?0:g>65534?65534:g;
        src.blue=b<0?0:b>65534?65534:b;

        dst.red= src.red&0xF800;
        dst.green= src.green&0xF800;
        dst.blue= src.blue&0xF800;

        (*table)[32768+index].pel[i]=(dst.red>>1)|(dst.green>>6)|(dst.blue>>11);

        err[32768+index].red=dst.red-src.red;
        err[32768+index].green=dst.green-src.green;
        err[32768+index].blue=dst.blue-src.blue;
    }
    DisposePtr((Ptr)err);
    return(noErr);
}

Boolean
GreyClut(CTabHandle clut)
{
    Boolean result=true;
    int     i, size;

    size=(*clut)->ctSize;
    for(i=0;i<=size && result;i++) {
        int     r,g,b;

        r=(*clut)->ctTable[i].rgb.red;
        g=(*clut)->ctTable[i].rgb.green;
        b=(*clut)->ctTable[i].rgb.blue;

        result=(r==g && g==b);
    }
```

Engineering:KlicsCode:CompPict:Clut.c

```
return result;
;
```

Engineering:KlicsCode:CompPict:Bits3.h

```
/*******************************
 *
 * © Copyright 1993 KLICS Limited
 * All rights reserved.
 *
 * Written by: Adrian Lewis
 *
 *******************************/
/*
    Bits3.h: fast bit read/write definitions

    buf_use     define static variables
    buf_winit   initialise vars for write
    buf_rinit   initialise vars for read
    buf_set     set current bit
    buf_get     get current bit
    buf_winc    increment write buffer
    buf_rinc    increment read buffer
    buf_size    fullness of buffer in bytes
    buf_flush   flush buffer

    User defined macro/function buf_over must be defined in case of buffer overflo
*/

typedef struct {
    unsigned long   *buf;
    union   {
        unsigned long   mask;
        long    bno;
    } index;
    unsigned long   *ptr, data, size;
} Buffer, *Buf;

#define buf_winit(buf) \
    buf->index.mask=0x80000000; \
    buf->ptr=&buf->buf[0]; \
    buf->data=0;

#define buf_rinit(buf) \
    buf->index.bno=0; \
    buf->ptr=&buf->buf[0];

#define buf_set(buf) \
    buf->data |= buf->index.mask;

#define buf_get(buf) \
    0!=(buf->data & (1<<buf->index.bno) )

#define buf_winc(buf) \
    if (buf->index.mask==1) { \
        *buf->ptr=buf->data; \
        buf->data=0; \
        buf->index.mask=0x80000000; \
        buf->ptr++; \
    } else buf->index.mask >>= 1;

#define buf_rinc(buf) \
    if (--(buf->index.bno)<0) { \
        buf->data=*buf->ptr++; \
        buf->index.bno=31; \
    };

/* buf_size only valid after buf_flush */
```

Engineering:KlicsCode:CompPict:Bits3.h

```
#define buf_size(buf) \
    (unsigned char *)buf->ptr-(unsigned char *)&buf->buf[0]

#define buf_flush(buf) \
    if (buf->index.mask!=0x80000000) { \
        buf->data|=buf->index.mask-1; \
        *buf->ptr=buf->data; \
        buf->ptr++; \
    }
```

Engineering:KlicsCode:CompPict:Bits3.a

```
*--------------------------------
*
*   © Copyright 1993 KLICS Limited
*   All rights reserved.
*
*   Written by: Adrian Lewis
*
*--------------------------------
*
*   68000 Bit buffer code (Bits2.h)
*
*--------------------------------
*
*   Macros:
*       buf_winit   &ptr,&data,&mask,&buf
*       buf_rinit   &ptr,&bno,&buf
*       buf_set     &data,&mask
*       buf_get     &data,&bno
*       buf_winc    &ptr,&data,&mask
*       buf_rinc    &ptr,&data,&index
*       buf_flush   &ptr,&data,&mask
*
*--------------------------------
        macro
        buf_winit   &ptr,&data,&mask,&buf
*
        move.l      #$80000000,&mask        ; mask=100..
        move.l      &buf,&ptr               ; ptr=buf
        clr.l       &data                   ; data=0
*
        endm
*--------------------------------
        macro
        buf_rinit   &ptr,&bno,&buf
*
        clr.b       &bno                    ; bno=0
        move.l      &buf,&ptr               ; ptr=buf
*
        endm
*--------------------------------
        macro
        buf_set     &data,&mask
*
        or.l        &mask,&data             ; data |= mask
*
        endm
*--------------------------------
        macro
        buf_get     &data,&bno
*
        subq.b      #1,&bno
        btst        &bno,&data
*
        endm
*--------------------------------
        macro
        buf_winc    &ptr,&data,&mask
*
        lsr.l       #1,&mask                ; mask>>=1
        bne.s       @cont                   ; if non-zero continue
        move.l      &data,(&ptr)+           ; *ptr++=data
        clr.l       &data                   ; data=0
        move.l      #$80000000,&mask        ; mask=100...
```

Engineering:KlicsCode:CompPict:Bits3.a

```
@cont
*
        endm
*-------------------------------------------
        macro
        buf_rinc    &ptr,&data,&bno
*
        cmpi.b      #16,&bno
        bge.s       @cont
        swap        &data
        move.w      (&ptr)+,&data           ; data=*ptr++
        add.b       #16,&bno                ; bno+=16
@cont
*
        endm
*-------------------------------------------
        macro
        buf_flush   &ptr,&data,&mask
*
        cmp.l       #$80000000,&mask        ; mask=80000000?
        beq.s       @cont                   ; if buffer empty continue
        move.l      &data,(&ptr)+           ; *ptr++=data
*
        endm
*-------------------------------------------
```

Engineering:KlicsCode:CompPict:Backward.c

```
/*****************************
 *
 *  © Copyright 1993 KLICS Limited
 *  All rights reserved.
 *
 *  Written by: Adrian Lewis
 *
 ******************************/
/*
    Extra fast Backward\convolver
    New wavelet coeffs : 3 5 1 1, 1 2 1, 1 1

    Optimized for speed:
        dirn - False
        src/dst octave == 0
*/

#define BwdS0(addr0,dAG,dAH,dBH) \
    v=*(short *)addr0; \
    dAG= -v; \
    dAH= v; \
    dBH= v<<1; \

#define BwdS1(addr1,addr0,dAG,dAH,dBH) \
    v=*(short *)addr1; \
    dBH+= v>>1; \
    dAG+= v+(vs=v<<1); \
    dAH-= v+(vs<<=1); \
    *(short *)addr0=dBH>>1;

#define Bwd2(addr2,dAG,dAH,dBG,dBH) \
    v=*(short *)addr2; \
    dBG= -v; \
    dBH= v; \
    dAH+= v+(vs=v<<1); \
    dAG+= v+(vs<<=1);

#define Bwd3(addr3,addr2,addr1,dAG,dAH,dBG,dBH) \
    v=*(short *)addr3; \
    dAH+= v; \
    dAG+= v; \
    dBG+= v+(vs=v<<1); \
    dBH-= v+(vs<<=1); \
    *(short *)addr1=(dAH+1)>>2; \
    *(short *)addr2=(dAG+1)>>2;

#define Bwd0(addr0,dAG,dAH,dBG,dBH) \
    v=*(short *)addr0; \
    dAG= -v; \
    dAH= v; \
    dBH+= v+(vs=v<<1); \
    dBG+= v+(vs<<=1);

#define Bwd1(addr1,addr0,addr3,dAG,dAH,dBG,dBH) \
    v=*(short *)addr1; \
    dBH+= v; \
    dBG+= v; \
    dAG+= v+(vs=v<<1); \
    dAH-= v+(vs<<=1); \
    *(short *)addr3=(dBH+1)>>2; \
    *(short *)addr0=(dBG+1)>>2;

#define BwdE2(addr2,dAG,dAH,dBH) \
```

```
    v=*(short *)addr2; \
    dBH= vs=v<<1; \
    dAH+= v-(vs=v<<1); \
    dAG+= v+(vs<<=1);

#define BwdE3(addr3,addr2,addr1,dAG,dAH,dBH) \
    v=*(short *)addr3; \
    dAH+= v; \
    dAG+= v; \
    dBH-= v+(vs=v<<1); \
    dBH-= v+(vs<<=1); \
    *(short *)addr1=(dAH+1)>>2; \
    *(short *)addr2=(dAG+1)>>2; \
    *(short *)addr3=dBH>>1;

#define Bwd(base,end,inc) \
    addr0=base; \
    addr3=addr0-(inc>>2); \
    addr2=addr3-(inc>>2); \
    addr1=addr2-(inc>>2); \
    BwdS0(addr0,dAG,dAH,dBH); \
    addr1+=inc; \
    BwdS1(addr1,addr0,dAG,dAH,dBH); \
    addr2+=inc; \
    while(addr2<end) { \
        Bwd2(addr2,dAG,dAH,dBG,dBH); \
        addr3+=inc; \
        Bwd3(addr3,addr2,addr1,dAG,dAH,dBG,dBH); \
        addr0+=inc; \
        Bwd0(addr0,dAG,dAH,dBG,dBH); \
        addr1+=inc; \
        Bwd1(addr1,addr0,addr3,dAG,dAH,dBG,dBH); \
        addr2+=inc; \
    } \
    BwdE2(addr2,dAG,dAH,dBH); \
    addr3+=inc; \
    BwdE3(addr3,addr2,addr1,dAG,dAH,dBH);

#define BwdS0r2(addr0,dAG,dAH,dBH) \
    v=*(short *)addr0; \
    dAG= 0; \
    dAH= v; \
    dBH= v; \

#define BwdS1r2(addr1,addr0,dAG,dAH,dBH) \
    v=*(short *)addr1; \
    dBH+= v>>2; \
    dAG+= v; \
    dAH-= v<<1; \
    *(short *)addr0=dBH;

#define Bwd2r2(addr2,dAG,dAH,dBG,dBH) \
    v=*(short *)addr2; \
    dBG= 0; \
    dBH= v; \
    dAH+= v; \
    dAG+= v<<1;

#define Bwd3r2(addr3,addr2,addr1,dAG,dAH,dBG,dBH) \
    v=*(short *)addr3; \
    dAH+= 0; \
    dAG+= v; \
    dBG+= v; \
```

```
    dBH-= v<<1; \
    *(short *)addr1=dAH>>1; \
    *(short *)addr2=dAG>>1;

#define Bwd0r2(addr0,dAG,dAH,dBG,dBH) \
    v=*(short *)addr0; \
    dAG=  0; \
    dAH=  v; \
    dBH+= v; \
    dBG-= v<<1;

#define Bwd1r2(addr1,addr0,addr3,dAG,dAH,dBG,dBH) \
    v=*(short *)addr1; \
    dBH+= 0; \
    dBG+= v; \
    dAG+= v; \
    dAH-= v<<1; \
    *(short *)addr3=dBH>>1; \
    *(short *)addr0=dBG>>1;

#define BwdE2r2(addr2,dAG,dAH,dBH) \
    v=*(short *)addr2; \
    dBH=  v; \
    dAH+= v; \
    dAG+= v<<1;

#define BwdE3r2(addr3,addr2,addr1,dAG,dAH,dBH) \
    v=*(short *)addr3; \
    dAH+= 0; \
    dAG+= v; \
    dBH-= v; \
    dBH-= v<<1; \
    *(short *)addr1=dAH>>1; \
    *(short *)addr2=dAG>>1; \
    *(short *)addr3=dBH;

#define Bwdr2(base,end,inc) \
    addr0=base; \
    addr3=addr0-(inc>>2); \
    addr2=addr3-(inc>>2); \
    addr1=addr2-(inc>>2); \
    BwdS0r2(addr0,dAG,dAH,dBH); \
    addr1+=inc; \
    BwdS1r2(addr1,addr0,dAG,dAH,dBH); \
    addr2+=inc; \
    while(addr2<end) { \
        Bwd2r2(addr2,dAG,dAH,dBG,dBH); \
        addr3+=inc; \
        Bwd3r2(addr3,addr2,addr1,dAG,dAH,dBG,dBH); \
        addr0+=inc; \
        Bwd0r2(addr0,dAG,dAH,dBG,dBH); \
        addr1+=inc; \
        Bwd1r2(addr1,addr0,addr3,dAG,dAH,dBG,dBH); \
        addr2+=inc; \
    } \
    BwdE2r2(addr2,dAG,dAH,dBH); \
    addr3+=inc; \
    BwdE3r2(addr3,addr2,addr1,dAG,dAH,dBH);

#define BwdS0r3(addr0,dAG,dAH,dBH) \
    v=*(short *)addr0; \
    dAG= 0; \
    dAH= 0; \
```

```
    dBH= v>>1; \

#define BwdS1r3(addr1,addr0,dAG,dAH,dBH) \
    v=*(short *)addr1; \
    dBH+= v>>3; \
    dAG+= v; \
    dAH-= v; \
    *(short *)addr0=dBH<<1;

#define Bwd2r3(addr2,dAG,dAH,dBG,dBH) \
    v=*(short *)addr2; \
    dBG= 0; \
    dBH= 0; \
    dAH+= v; \
    dAG+= v;

#define Bwd3r3(addr3,addr2,addr1,dAG,dAH,dBG,dBH) \
    v=*(short *)addr3; \
    dAH+= 0; \
    dAG+= 0; \
    dBG+= v; \
    dBH-= v; \
    *(short *)addr1=dAH; \
    *(short *)addr2=dAG;

#define Bwd0r3(addr0,dAG,dAH,dBG,dBH) \
    v=*(short *)addr0; \
    dAG= 0; \
    dAH= 0; \
    dBH+= v; \
    dBG+= v;

#define Bwd1r3(addr1,addr0,addr3,dAG,dAH,dBG,dBH) \
    v=*(short *)addr1; \
    dBH+= 0; \
    dBG+= 0; \
    dAG+= v; \
    dAH-= v; \
    *(short *)addr3=dBH; \
    *(short *)addr0=dBG;

#define BwdE2r3(addr2,dAG,dAH,dBH) \
    v=*(short *)addr2; \
    dBH= v>>1; \
    dAH+= v; \
    dAG+= v;

#define BwdE3r3(addr3,addr2,addr1,dAG,dAH,dBH) \
    v=*(short *)addr3; \
    dAH+= 0; \
    dAG+= 0; \
    dBH-= v; \
    dBH-= v; \
    *(short *)addr1=dAH; \
    *(short *)addr2=dAG; \
    *(short *)addr3=dBH<<1;

#define Bwdr3(base,end,inc) \
    addr0=base; \
    addr3=addr0-(inc>>2); \
    addr2=addr3-(inc>>2); \
    addr1=addr2-(inc>>2); \
    BwdS0r3(addr0,dAG,dAH,dBH); \
```

```
    addr1+=inc; \
    BwdS1r3(addr1,addr0,dAG,dAH,dBH); \
    addr2+=inc; \
    while(addr2<end) { \
        Bwd2r3(addr2,dAG,dAH,dBG,dBH); \
        addr3-=inc; \
        Bwd3r3(addr3,addr2,addr1,dAG,dAH,dBG,dBH); \
        addr0+=inc; \
        Bwd0r3(addr0,dAG,dAH,dBG,dBH); \
        addr1+=inc; \
        Bwd1r3(addr1,addr0,addr3,dAG,dAH,dBG,dBH); \
        addr2+=inc; \
    } \
    BwdE2r3(addr2,dAG,dAH,dBH); \
    addr3+=inc; \
    BwdE3r3(addr3,addr2,addr1,dAG,dAH,dBH);

extern void FASTBACKWARD(char *data, long inc1, long loop1, long inc2, char *end2)
extern void HAARBACKWARD(char *data, long inc1, long loop1, long inc2, long loop2)
extern void HAARTOPBWD(char *data,long height,long width);
/* extern void  HAARXTOPBWD(char *data,long area);*/

void    FasterBackward(char *data, long inc1, long end1, long inc2, char *end2)
{
    register short  v, vs, v3, dAG, dAH, dBG, dBH, inc;
    register char   *addr0, *addr1, *addr2, *addr3, *end;
    char    *base;

    inc=inc1;
    for(base=data;base<end2;base+=inc2) {
        end=base+end1;
        Bwd(base,end,inc);
    }
}

extern  void    TOPBWD(char *data, char *dst, long size_1, long size_0);

void    TestTopBackward(short *data,int size[2],int oct_src)
{
    int     oct, area=size[0]*size[1]<<1;
    short   width=size[0]<<1;
    char    *top=area+(char *)data, *left=width+(char *)data;

    for(oct=oct_src-1;oct>0;oct--) {
        long    cinc=2<<oct, cinc4=cinc<<2,
                rinc=size[0]<<oct+1, rinc4=rinc<<2; /* col and row increments in t

        FASTBACKWARD((char *)data,rinc4,area-(rinc<<1),cinc,left);
        FASTBACKWARD((char *)data,cinc4,width-(cinc<<1),rinc,top);
    }
/*  FasterBackward((char *)data,size[0]<<3,area-(size[0]<<2),2,left);
    FasterBackward((char *)data,8,width-4,size[0]<<1,top);*/
    TOPBWD((char *)data,(char *)data,size[0],size[1]);
}
void    TestBackward(data,size,oct_src)

short   *data;
int     size[2], oct_src;

{
    int     oct, area=size[0]*size[1]<<1;
    short   width=size[0]<<1;
    char    *top=area+(char *)data, *left=width+(char *)data;
```

Engineering:KlicsCode:CompPict:Backward.c

```
    for(oct=oct_src-1;oct>=0;oct--) {
        long    cinc=2<<oct, cinc4=cinc<<2,
                rinc=size[0]<<oct+1, rinc4=rinc<<2; /* col and row increments in t

        FasterBackward((char *)data,rinc4,area-(rinc<<1),cinc,left);
        FasterBackward((char *)data,cinc4,width-(cinc<<1),rinc,top);
    }
}

void    Backward3511(data,size,oct_src)

short   *data;
int     size[2], oct_src;

{
    int     oct, area=size[0]*size[1]<<1;
    short   width=size[0]<<1;
    char    *top=area+(char *)data, *left=width+(char *)data;

    for(oct=oct_src-1;oct>0;oct--) {
        long    cinc=2<<oct, cinc4=cinc<<2,
                rinc=size[0]<<oct+1, rinc4=rinc<<2; /* col and row increments in t


        BACK3511((char *)data,rinc4,area-(rinc<<1),cinc,left);
        BACK3511((char *)data,cinc4,width-(cinc<<1),rinc,top);
    }
    BACK3511V((char *)data,size[0]<<3,area-(size[0]<<2),4,left);
    BACK3511H((char *)data,8,width-4,size[0]<<1,top);
/*  TOPBWD((char *)data,(char *)data,size[1],size[0]);*/
}
```

Engineering:KlicsCode:CompPict:Backward.a

```
*-------------------------------
*
*   © Copyright 1993 KLICS Limited
*   All rights reserved.
*
*   Written by: Adrian Lewis
*
*-------------------------------
*
*   680X0 3511 Backward code
*
*   Coeffs 11 19 5 3
*   become  3  5 1 1
*
*-------------------------------
        seg         'klics'
*-------------------------------
        macro
        BwdStart0   &addr0,&dAG,&dAH,&dBH
*
        move.w      (&addr0),&dAH   ; dAH=*(short *)addr0
        move.w      &dAH,&dAG       ; dAG=v
        neg.w       &dAG            ; dAG= -dAG
        move.w      &dAH,&dBH       ; dBH=v
        add.w       &dBH,&dBH       ; dBH=v<<1
*
        endm
*-------------------------------
        macro
        BwdStart1   &addr1,&addr0,&dAG,&dAH,&dBH
*
        move.w      (&addr1),d0     ; v=*(short *)addr1
        move.w      d0,d1           ; vs=v
        asr.w       #1,d1           ; vs=v>>1
        add.w       d1,&dBH         ; dBH+= v>>1
        add.w       d0,&dAG         ; dAG+=v
        sub.w       d0,&dAH         ; dAH-=v
        add.w       d0,d0           ; v<<=1
        add.w       d0,&dAG         ; dAG+=2v
        add.w       d0,d0           ; v<<=1
        sub.w       d0,&dAH         ; dAH-=4v
        asr.w       #1,&dBH         ; dBH>>=1
        move.w      &dBH,(&addr0)   ; *(short *)addr0=dBH
*
        endm
*-------------------------------
        macro
        BwdEven &addr2,&dAG,&dAH,&dBG,&dBH
*
        move.w      (&addr2),d0     ; v=*(short *)addr2
        move.w      d0,&dBH         ; dBH=v
        move.w      d0,&dBG         ; dBG=v
        neg.w       &dBG            ; dBG=-v
        add.w       d0,&dAH         ; dAH+=v
        add.w       d0,&dAG         ; dAG+=v
        add.w       d0,d0           ; 2v
        add.w       d0,&dAH         ; dAH+=v
        add.w       d0,d0           ; 2v
        add.w       d0,&dAG         ; dAH+=v
*
        endm
*-------------------------------
        macro
```

Engineering:KlicsCode:CompPict:Backward.a

```
        BwdOdd      &addr3,&addr2,&addr1,&dAG,&dAH,&dBG,&dBH
*
        move.w      (&addr3),d0       ; v=*(short *)addr3

        add.w       d0,&dAH           ; dAH+=v
        add.w       d0,&dAG           ; dAG+=v
        add.w       d0,&dBG           ; dBG+=v
        sub.w       d0,&dBH           ; dBH-=v
        add.w       d0,d0             ; 2v
        add.w       d0,&dBG           ; dBG+=v
        add.w       d0,d0             ; 4v
        sub.w       d0,&dBH           ; dBH-=4v

        asr.w       #2,&dAH           ; dAH>>=2
        move.w      &dAH,(&addr1)     ; *(short *)addr1=dAH
        asr.w       #2,&dAG           ; dAG>>=2
        move.w      &dAG,(&addr2)     ; *(short *)addr2=dAG
*
        endm
*---------------------------------
        macro
        BwdEnd2     &addr2,&dAG,&dAH,&dBH
*
        move.w      (&addr2),d0       ; v=*(short *)addr2

        add.w       d0,&dAH           ; dAH+=v
        add.w       d0,&dAG           ; dAG+=v
        add.w       d0,d0             ; 2v
        move.w      d0,&dBH           ; dBH=2v
        add.w       d0,&dAH           ; dAH+=2v
        add.w       d0,d0             ; 4v
        add.w       d0,&dAG           ; dAG+=4v
*
        endm
*---------------------------------
        macro
        BwdEnd3     &addr3,&addr2,&addr1,&dAG,&dAH,&dBH
*
        move.w      (&addr3),d0       ; v=*(short *)addr3
        add.w       d0,&dAH           ; dAH+=v
        add.w       d0,&dAG           ; dAG+=v
        lsl.w       #3,d0             ; 8v
        sub.w       d0,&dBH           ; dBH-=8v
        asr.w       #2,&dAH           ; dAH>>=2
        move.w      &dAH,(&addr1)     ; *(short *)addr1=dAH
        asr.w       #2,&dAG           ; dAG>>=2
        move.w      &dAG,(&addr2)     ; *(short *)addr2=dAG
        asr.w       #1,&dBH           ; dBH>>=1
        move.w      &dBH,(&addr3)     ; *(short *)addr3=dBH
*
        endm
*---------------------------------
        macro
        Bwd         &base,&end,&inc
*
        movea.l     &base,a0                ; addr0=base
        move.l      &inc,d0                 ; d0=inc
        asr.l       #2,d0                   ; d0=inc>>2
        movea.l     a0,a3                   ; addr3=addr0
        suba.l      d0,a3                   ; addr3-=(inc>>2)
        movea.l     a3,a2                   ; addr2=addr3
        suba.l      d0,a2                   ; addr2-=(inc>>2)
        movea.l     a2,a1                   ; addr1=addr2
```

Engineering:KlicsCode:CompPict:Backward.a

```
        suba.l      d0,a1                       ; addr1-=(inc>>2)
        BwdStart0   a0,d4,d5,d7                 ; BwdStart0(addr0,dAG,dAH,dBH)
        adda.l      &inc,a1                     ; addr1+=inc
        BwdStart1   a1,a0,d4,d5,d7              ; BwdStart1(addr1,addr0,dAG,dAH,dBH)
        adda.l      &inc,a2                     ; addr2+=inc
@do     BwdEven     a2,d4,d5,d6,d7              ; BwdEven(addr2,dAG,dAH,dBG,dBH)
        adda.l      &inc,a3                     ; addr3+=inc
        BwdOdd      a3,a2,a1,d4,d5,d6,d7        ; BwdOdd(addr3,addr2,addr1,dAG,dAH,dBG
        adda.l      &inc,a0                     ; addr0+=inc
        BwdEven     a0,d6,d7,d4,d5              ; BwdEven(addr0,dBG,dBH,dAG,dAH)
        adda.l      &inc,a1                     ; addr1+=inc
        BwdOdd      a1,a0,a3,d6,d7,d4,d5        ; BwdOdd(addr1,addr0,addr3,dBG,dBH,dAG
        adda.l      &inc,a2                     ; addr2+=inc
        cmpa.l      a2,&end                     ; addr2<end
        bgt.s       @do                         ; while
        BwdEnd2     a2,d4,d5,d7                 ; BwdEnd2(addr2,dAG,dAH,dBH)
        adda.l      &inc,a3                     ; addr3+=inc
        BwdEnd3     a3,a2,a1,d4,d5,d7           ; BwdEnd3(addr3,addr2,addr1,dAG,dAH,dB
*
        endm
*-----------------------------------
Back3511    FUNC    EXPORT
*
PS      RECORD      8
data    DS.L        1
inc1    DS.L        1
end1    DS.L        1
inc2    DS.L        1
end2    DS.L        1
        ENDR
*
        link        a6,#0                       ; no local variables
        movem.l     d4-d7/a3-a5,-(a7)           ; store registers
*
        move.l      PS.inc1(a6),d3              ; inc=inc1
        movea.l     PS.data(a6),a5              ; base=data
@do     movea.l     a5,a4                       ; end=base
        adda.l      PS.end1(a6),a4              ; end+=end1
        Bwd         a5,a4,d3                    ; Bwd(base,end,inc)
        adda.l      PS.inc2(a6),a5              ; base+=inc2
        cmpa.l      PS.end2(a6),a5              ; end2>base
        blt.w       @do                         ; for
*
        movem.l     (a7)+,d4-d7/a3-a5           ; restore registers
        unlk        a6                          ; remove locals
        rts                                     ; return
*
        ENDFUNC
*-----------------------------------
        macro
        BwdStartV0  &addr0,&dAG,&dAH,&dBH
*
        move.l      (&addr0),&dAH   ; dAH=*(short *)addr0
        move.l      &dAH,&dAG       ; dAG=v
        neg.l       &dAG            ; dAG= -dAG
        move.l      &dAH,&dBH       ; dBH=v
        add.l       &dBH,&dBH       ; dBH=v<<1
*
        endm
*-----------------------------------
        macro
        BwdStartV1  &addr1,&addr0,&dAG,&dAH,&dBH
*
```

≠Engineering:KlicsCode:CompPict:Backward.a

```
        move.l      (&addr1),d0         ; v=*(short *)addr1
        move.l      d0,d1               ; vs=v
        asr.l       #1,d1               ; vs=v>>1
        add.l       d1,&dBH             ; dBH+= v>>1
        add.l       d0,&dAG             ; dAG+=v
        sub.l       d0,&dAH             ; dAH-=v
        add.l       d0,d0               ; v<<=1
        add.l       d0,&dAG             ; dAG+=2v
        add.l       d0,d0               ; v<<=1
        sub.l       d0,&dAH             ; dAH-=4v

        asr.l       #1,&dBH             ; dBH>>=1
        add.w       &dBH,&dBH           ; shift word back
        asr.w       #1,&dBH             ; dBH>>=1
        move.l      &dBH,(&addr0)       ; *(short *)addr0=dBH
*
        endm
*-----------------------------------
        macro
        BwdEvenV    &addr2,&dAG,&dAH,&dBG,&dBH
*
        move.l      (&addr2),d0         ; v=*(short *)addr2
        move.l      d0,&dBH             ; dBH=v
        move.l      d0,&dBG             ; dBG=v
        neg.l       &dBG                ; dBG=-v
        add.l       d0,&dAH             ; dAH+=v
        add.l       d0,&dAG             ; dAG+=v
        add.l       d0,d0               ; 2v
        add.l       d0,&dAH             ; dAH+=v
        add.l       d0,d0               ; 2v
        add.l       d0,&dAG             ; dAH+=v
*
        endm
*-----------------------------------
        macro
        BwdOddV     &addr3,&addr2,&addr1,&dAG,&dAH,&dBG,&dBH
*
        move.l      (&addr3),d0         ; v=*(short *)addr3

        add.l       d0,&dAH             ; dAH+=v
        add.l       d0,&dAG             ; dAG+=v
        add.l       d0,&dBG             ; dBG+=v
        sub.l       d0,&dBH             ; dBH-=v
        add.l       d0,d0               ; 2v
        add.l       d0,&dBG             ; dBG+=v
        add.l       d0,d0               ; 4v
        sub.l       d0,&dBH             ; dBH-=4v

        asr.l       #2,&dAH             ; dAH>>=2
        lsl.w       #2,&dAH             ; shift word back
        asr.w       #2,&dAH             ; dAH>>=2
        move.l      &dAH,(&addr1)       ; *(short *)addr1=dAH
        asr.l       #2,&dAG             ; dAG>>=2
        lsl.w       #2,&dAG             ; shift word back
        asr.w       #2,&dAG             ; dAG>>=2
        move.l      &dAG,(&addr2)       ; *(short *)addr2=dAG
*
        endm
*-----------------------------------
        macro
        BwdEndV2    &addr2,&dAG,&dAH,&dBH
*
        move.l      (&addr2),d0         ; v=*(short *)addr2
```

Engineering:KlicsCode:CompPict:Backward.a

```
	add.l		d0,&dAH		; dAH+=v
	add.l		d0,&dAG		; dAG+=v
	add.l		d0,d0		; 2v
	move.l		d0,&dBH		; dBH=2v
	add.l		d0,&dAH		; dAH+=2v
	add.l		d0,d0		; 4v
	add.l		d0,&dAG		; dAG+=4v
*
	endm
*--------------------------------
	macro
	BwdEndV3	&addr3,&addr2,&addr1,&dAG,&dAH,&dBH
*
	move.l		(&addr3),d0	; v=*(short *)addr3
	add.l		d0,&dAH		; dAH+=v
	add.l		d0,&dAG		; dAG+=v
	lsl.l		#3,d0		; 8v
	sub.l		d0,&dBH		; dBH-=8v
	asr.l		#2,&dAH		; dAH>>=2
	lsl.w		#2,&dAH		; shift word back
	asr.w		#2,&dAH		; dAH>>=2
	move.l		&dAH,(&addr1)	; *(short *)addr1=dAH
	asr.l		#2,&dAG		; dAG>>=2
	lsl.w		#2,&dAG		; shift word back
	asr.w		#2,&dAG		; dAG>>=2
	move.l		&dAG,(&addr2)	; *(short *)addr2=dAG
	asr.l		#1,&dBH		; dBH>>=1
	lsl.w		#1,&dBH		; shift word back
	asr.w		#1,&dBH		; dAH>>=2
*	add.l		&dBH,&dBH	; dBH<<=1
	move.l		&dBH,(&addr3)	; *(short *)addr3=dBH
*
	endm
*--------------------------------
	macro
	BwdV		&base,&end,&inc
*
	movea.l		&base,a0		; addr0=base
	move.l		&inc,d0			; d0=inc
	asr.l		#2,d0			; d0=inc>>2
	movea.l		a0,a3			; addr3=addr0
	suba.l		d0,a3			; addr3-=(inc>>2)
	movea.l		a3,a2			; addr2=addr3
	suba.l		d0,a2			; addr2-=(inc>>2)
	movea.l		a2,a1			; addr1=addr2
	suba.l		d0,a1			; addr1-=(inc>>2)
	BwdStartV0	a0,d4,d5,d7		; BwdStart0(addr0,dAG,dAH,dBH)
	adda.l		&inc,a1			; addr1+=inc
	BwdStartV1	a1,a0,d4,d5,d7		; BwdStart1(addr1,addr0,dAG,dAH,dBH)
	adda.l		&inc,a2			; addr2+=inc
@do	BwdEvenV	a2,d4,d5,d6,d7		; BwdEven(addr2,dAG,dAH,dBG,dBH)
	adda.l		&inc,a3			; addr3+=inc
	BwdOddV		a3,a2,a1,d4,d5,d6,d7	; BwdOdd(addr3,addr2,addr1,dAG,dAH,dBG
	adda.l		&inc,a0			; addr0+=inc
	BwdEvenV	a0,d6,d7,d4,d5		; BwdEven(addr0,dBG,dBH,dAG,dAH)
	adda.l		&inc,a1			; addr1+=inc
	BwdOddV		a1,a0,a3,d6,d7,d4,d5	; BwdOdd(addr1,addr0,addr3,dBG,dBH,dAG
	adda.l		&inc,a2			; addr2+=inc
	cmpa.l		a2,&end			; addr2<end
	bgt.s		@do			; while
	BwdEndV2	a2,d4,d5,d7		; BwdEnd2(addr2,dAG,dAH,dBH)
	adda.l		&inc,a3			; addr3+=inc
```

Engineering:KlicsCode:CompPict:Backward.a

```
        BwdEndV3        a3,a2,a1,d4,d5,d7          ; BwdEnd3(addr3,addr2,addr1,dAG,dAH,dB
*
        endm
*--------------------------------
Back3511V   FUNC        EXPORT
*
PS      RECORD          8
data    DS.L            1
incl    DS.L            1
end1    DS.L            1
inc2    DS.L            1
end2    DS.L            1
        ENDR
*
        link            a6,#0                      ; no local variables
        movem.l         d4-d7/a3-a5,-(a7)          ; store registers
*
        move.l          PS.incl(a6),d3             ; inc=incl
        movea.l         PS.data(a6),a5             ; base=data
@do     movea.l         a5,a4                      ; end=base
        adda.l          PS.end1(a6),a4             ; end+=end1
        BwdV            a5,a4,d3                   ; Bwd(base,end,inc)
        adda.l          PS.inc2(a6),a5             ; base+=inc2
        cmpa.l          PS.end2(a6),a5             ; end2>base
        blt.w           @do                        ; for
*
        movem.l         (a7)+,d4-d7/a3-a5          ; restore registers
        unlk            a6                         ; remove locals
        rts                                        ; return
*
        ENDFUNC
*--------------------------------
        macro
        BwdStartH       &addrR,&A,&C
*
        move.l          (&addrR)+,&A       ; 1H1G=*(long *)addrR
        move.l          &A,d0              ; A=1H1G, d0=1H1G
        move.l          &A,&C              ; A=1H1G, d0=1H1G, C=1H1G
        add.w           &A,d0              ; A=1H1G, d0=1H2G, C=1H1G
        add.w           d0,&A              ; A=1H3G, d0=1H2G, C=1H1G
        add.w           &A,d0              ; A=1H3G, d0=1H5G, C=1H1G
        swap            &A                 ; A=3GH1, d0=1H5G, C=1H1G
        sub.l           d0,&A              ; A=AAAA, d0=1H5G, C=1H1G
*
        endm
*--------------------------------
        macro
        BwdCycleH       &addrR,&addrW,&A,&B,&C
*
        move.l          (&addrR)+,&B       ; 1H1G=*(long *)addrR
        move.l          &B,d0              ; B=1H1G, d0=1H1G
        add.l           d0,d0              ; B=1H1G, d0=2H2G
        move.l          d0,d1              ; B=1H1G, d0=2H2G, d1=2H2G
        add.l           &B,d0              ; B=1H1G, d0=3H3G, d1=2H2G
        add.l           d0,d1              ; B=1H1G, d0=3H3G, d1=5H5G
        move.l          &B,d2              ; B=1H1G, d0=3H3G, d1=5H5G, d2=1H1G
        move.w          d1,d2              ; B=1H1G, d0=3H3G, d1=5H5G, d2=1H5G
        move.w          &B,d1              ; B=1H1G, d0=3H3G, d1=5H1G, d2=1H5G
        move.w          d0,&B              ; B=1H3G, d0=3H3G, d1=5H1G, d2=1H5G
        move.w          d1,d0              ; B=1H3G, d0=3H1G, d1=5H1G, d2=1H5G
        swap            &B                 ; B=3G1H, d0=3H1G, d1=5H1G, d2=1H5G
        swap            d0                 ; B=3G1H, d0=1G3H, d1=5H1G, d2=1H5G
```

Engineering:KlicsCode:CompPict:Backward.a

```
        sub.l       d2,&B               ; B=3G1H-1H5G
        add.l       d0,&A               ; A+=1H3G
        add.l       d1,&A               ; A+=5G1H

        asr.w       #2,&A               ; A0>>=2
        move.w      &A,&C               ; C complete
        asr.l       #2,&A               ; A1>>=2
        move.l      &C,(&addrW)+        ; *(long *)addrW=DD
        move.l      &A,&C               ; C=A1XX
*
        endm
*-----------------------------------
        macro
        BwdEndH     &addrR,&addrW,&A,&B,&C
*
        move.l      (&addrR)+,d0        ; 1H1G=*(long *)addrR
        move.w      d0,d2               ; d2=1G
        lsl.w       #2,d2               ; d2=4G
        neg.w       d2                  ; d2=-4G
        swap        d0                  ; d0=1G1H
        add.w       d0,d2               ; d2+=1H
        move.l      d0,d1               ; d0=1G1H, d1=1G1H
        add.w       d0,d1               ; d0=1G1H, d1=1G2H
        add.w       d1,d0               ; d0=1G3H, d1=1G2H
        add.w       d0,d1               ; d0=1G3H, d1=1G5H
        swap        d1                  ; d0=1G3H, d1=5H1G
        add.l       d0,&A               ; A+=1G3H
        add.l       d1,&A               ; A+=5H1G

        asr.w       #2,&A               ; A1>>=2
        move.w      &A,&C               ; C complete
        asr.l       #2,&A               ; A0>>=2
        move.l      &C,(&addrW)+        ; *(long *)addrW=C
        move.w      d2,&A               ; A=D1D2
        move.l      &A,(&addrW)+        ; *(long *)addrW=A
*
        endm
*-----------------------------------
        macro
        BwdH        &base,&end,&inc
*
        movea.l     &base,a0                    ; addrR=base
        movea.l     a0,a1                       ; addrW=addrR
        BwdStartH   a0,d3,d5                    ; BwdStart(addrR,A,DD)
@do     BwdCycleH   a0,a1,d3,d4,d5              ; BwdCycle(addrR,addrW,A,B,C)
        BwdCycleH   a0,a1,d4,d3,d5              ; BwdCycle(addrR,addrW,B,A,C)
        cmpa.l      a0,&end                     ; addr2<end
        bgt.s       @do                         ; while
        BwdEndH     a0,a1,d3,d4,d5              ; BwdEnd(addrR,addrW,A,B,DD)
*
        endm
*-----------------------------------
Back3511H   FUNC    EXPORT
*
PS          RECORD  8
data        DS.L    1
inc1        DS.L    1
end1        DS.L    1
inc2        DS.L    1
end2        DS.L    1
            ENDR
*
            link    a6,#0                       ; no local variables
```

```
        movem.l     d4-d7/a3-a5,-(a7)       ; store registers
;
        move.l      PS.inc1(a6),d3          ; inc=inc1
        movea.l     PS.data(a6),a5          ; base=data
@do     movea.l     a5,a4                   ; end=base
        adda.l      PS.end1(a6),a4          ; end+=end1
        BwdH        a5,a4,d3                ; Bwd(base,end,inc)
        adda.l      PS.inc2(a6),a5          ; base+=inc2
        cmpa.l      PS.end2(a6),a5          ; end2>base
        blt.w       @do                     ; for
;
        movem.l     (a7)+,d4-d7/a3-a5       ; restore registers
        unlk        a6                      ; remove locals
        rts                                 ; return
;
        ENDFUNC
;-------------------------------
        END
```

```
/********************************
 *
 * © Copyright 1993 KLICS Limited
 * All rights reserved.
 *
 * Written by: Adrian Lewis
 *
 ********************************/
/*
 * Full still/video Knowles-Lewis Image KlicsEncode System utilising HVS propert:
 * and delta-tree coding
 *
 * Recoded and re-rationalised (Stand alone version)
 */

#include    <PixMath.h>
#include    "Bits3.h"
#include    "Klics.h"
#include    "KlicsHeader.h"
#include    "KlicsEncode.h"


#include    <Math.h>

/* If bool true the negate value */
#define negif(bool,value)   ((bool)?-(value):(value))
#define abs(value)          negif(value<0,value)

extern  void    HaarForward();
extern  void    Daub4Forward();

/* Use the bit level file macros (Bits2.h)
buf_use;*/

/* Huffman encode a block */
#define HuffEncLev(lev,buf) \
    HuffEncode(lev[0],buf); \
    HuffEncode(lev[1],buf); \
    HuffEncode(lev[2],buf); \
    HuffEncode(lev[3],buf);

/* Fixed length encode block of integers */
#define IntEncLev(lev,lpf_bits,buf) \
    IntEncode(lev[0],lpf_bits,buf); \
    IntEncode(lev[1],lpf_bits,buf); \
    IntEncode(lev[2],lpf_bits,buf); \
    IntEncode(lev[3],lpf_bits,buf);

/* Define write a zero */
#define Token0 \
    buf_winc(buf);

/* Define write a one */
#define Token1 \
    buf_set(buf); buf_winc(buf);

/* Write block for data and update memory */
#define DoXfer(addr,pro,lev,dst,mode,oct,nmode,buf) \
    HuffEncLev(lev,buf); \
    PutData(addr,pro,dst); \
    mode[oct]=oct==0?M_STOP:nmode;

/*  Function Name:  Quantize
```

Engineering:KlicsCode:CompPict:KlicsEnc.c

```
 *  Description:    H.261 style quantizer
 *  Arguments:  new, old - image blocks
 *              pro, lev - returned values
 *              q - quantizing divisor
 *  Returns:    lev is all zero, quantized data (pro) & level (lev)
 */

Boolean Quantize(int new[4], int old[4], int pro[4], int lev[4], short q)
{
    int     blk, half_q=(1<<q)-1>>1;

    for(blk=0;blk<4;blk++) {
        int     data=new[blk]-old[blk],
                mag_level=abs(data)>>q;

        mag_level=mag_level>135?135:mag_level;
        lev[blk]=negif(data<0,mag_level);
        pro[blk]=old[blk]+negif(data<0,(mag_level<<q)+(mag_level!=0?half_q:0));
    }
    return(pro[0]==0 && pro[1]==0 && pro[2]==0 && pro[3]==0);
}

void    QuantizeLPF(int new[4],int pro[4],int lev[4],short q)
{
    int     blk, half_q=(1<<q)-1>>1;

    for(blk=0;blk<4;blk++) {
        int     data=new[blk],
                mag_level=abs(data)>>q;

        lev[blk]=negif(data<0,mag_level);
        pro[blk]=(lev[blk]<<q)+half_q;
    }
}

/*  Function Name:  GuessQuantize
 *  Description:    Estimate threshold quantiser value
 *  Arguments:  new, old - image blocks
 *              q - q weighting factor
 *  Returns:    estimated q_const
 */

float   GuessQuantize(int new[4],int old[4],float q)
{
    int     blk;
    float   qt_max=0.0;

    for(blk=0;blk<4;blk++) {
        int     i, data=abs(new[blk]-old[blk]);
        float   qt;

        for(i=0;data!=0;i++) data>>=1;
        if (i>0) i--;
        qt=(((3<<i)-1)>>1)/q;

        qt_max=qt_max>qt?qt_max:qt;
    }
    return(qt_max);
}

/*  Function Name:  IntEncode
 *  Description:    Write a integer to bit file
 *  Arguments:  lev - integer to write now signed
```

```
 *              bits - no of bits
 */

void    IntEncode(int lev,int bits,Buf buf)
{
/* Old version
    int     i;

    for(i=bits-1;i>=0;i--) {
        if (lev&(1<<i)) buf_set(buf);
        buf_winc(buf);
    }
*/
/* New version
    int     i, mag=abs(lev);
    Boolean sign=lev<0;

    if (1<<bits-1 <= mag) mag=(1<<bits-1)-1;
    if (sign) buf_set(buf);
    buf_winc(buf);
    for(i=1<<bits-2;i!=0;i>>=1) {
        if (mag&i) buf_set(buf);
        buf_winc(buf);
    }*/
/* Hardware compatable version: sign mag(lsb->msb) */
    int     i, mag=abs(lev);
    Boolean sign=lev<0;

    if (1<<bits-1 <= mag) mag=(1<<bits-1)-1;
    if (sign) buf_set(buf);
    buf_winc(buf);
    for(i=1;i!=1<<bits-1;i<<=1) {
        if (mag&i) buf_set(buf);
        buf_winc(buf);
    }
}

/*  Function Name:  HuffEncodeSA
 *  Description:    Write a Huffman coded integer to bit file
 *  Arguments:  lev - integer value
 *  Returns:    no of bits used
 */

void    HuffEncode(int lev,Buf buf)
{
/*  int     level=abs(lev);

    if (level>1) buf_set(buf);
    buf_winc(buf);
    if(level>2 || level==1) buf_set(buf);
    buf_winc(buf);
    if (level!=0) {
        if (lev<0) buf_set(buf);
        buf_winc(buf);
        if (level>2) {
            int     i;

            for(i=3;i<level;i++) {
                buf_winc(buf);
            }
            buf_set(buf);
            buf_winc(buf);
        }
```

Engineering:KlicsCode:CompPict:KlicsEnc.c

```
    )*/

/* New version */
    int      level=abs(lev), i;

    if (level!=0) buf_set(buf);
    buf_winc(buf);
    if (level!=0) {
        if (lev<0) buf_set(buf);
        buf_winc(buf);
        if (level<8) {
            while (1<level--)
                buf_winc(buf);
            buf_set(buf);
            buf_winc(buf);
        } else {
            for(i=0;i<7;i++)
                buf_winc(buf);
            level-=8;
            for(i=1<<6;i!=0;i>>=1) {
                if (level&i) buf_set(buf);
                buf_winc(buf);
            }
        }
    }
}

/*  Function Name:  KlicsEChannel
 *  Description:    Encode a channel of image
 *  Arguments:  src - source channel memory
 *              dst - destination memory (and old for videos)
 *              octs, size - octaves of decomposition and image dimensions
 *              normals - HVS weighted normals
 *              lpf_bits - no of bits for LPF integer (image coding only)
 */

void    KlicsEncY(short *src,short *dst,int octs,int size[2],int thresh[5], int co
{
    int      oct, mask, x, y, sub, tmp, step=2<<octs, blk[4], mode[4], nz, no, base
    int      addr[4], new[4], old[4], pro[4], lev[4], zero[4]={0,0,0,0};
    Boolean nzflag, noflag, origin;
    int      bitmask=-1<<kle->seqh.precision-kle->frmh.quantizer[0]-1;
    Buf      buf=&kle->buf;

    for(y=0;y<size[1];y+=step)
    for(x=0;x<size[0];x+=step)
    for(sub=0;sub<4;sub++) {
    mode[oct=octs-1]=base_mode;
    if (sub==0) mode[oct=octs-1] |= M_LPF;
    mask=2<<oct;
    do {
        GetAddr(addr,x,y,sub,oct,size,mask);
        switch(mode[oct]) {
        case M_VOID:
            GetData(addr,old,dst);
            if (BlkZero(old)) mode[oct]=M_STOP;
            else { DoZero(addr,dst,mode,oct); }
            break;
        case M_SEND|M_STILL:
            GetData(addr,new,src);
            nz=Decide(new); nzflag=nz<=thresh[octs-oct];
            if (nzflag || Quantize(new,zero,pro,lev,kle->frmh.quantizer[octs-oct]))
                GetData(addr,old,dst);
```

Engineering:KlicsCode:CompPict:KlicsEnc.c

```
            if (BlkZero(old)) {
                Token0;
                mode[oct]=M_STOP;
            } else {
                Token1; Token1;
                DoZero(addr,dst,mode,oct);
            }
        } else {
            Token1; Token0;
            DoXfer(addr,pro,lev,dst,mode,oct,M_SEND|M_STILL,buf);
        }
        break;
    case M_SEND:
        GetData(addr,new,src);
        GetData(addr,old,dst);
        nz=Decide(new); nzflag=nz<=thresh[octs-oct];
        if (BlkZero(old)) {
            if (nzflag || Quantize(new,zero,pro,lev,kle->frmh.quantizer[octs-o
                Token0;
                mode[oct]=M_STOP;
            } else {
                Token1; Token0;
                DoXfer(addr,pro,lev,dst,mode,oct,M_SEND|M_STILL,buf);
            }
        } else {
            int     oz=Decide(old), no=DecideDelta(new,old);
            Boolean motion=(nz+oz)>>oct <= no; /* motion detection */

            no=DecideDelta(new,old); noflag=no<=compare[octs-oct];
            origin=nz<=no;
                if ((!noflag || motion)  && !nzflag) { /* was !noflag && !nzfl
                if (Quantize(new,origin?zero:old,pro,lev,kle->frmh.quantizer[o
                    Token1; Token1; Token0;
                    DoZero(addr,dst,mode,oct);
                } else {
                    if (origin) {
                        Token1; Token0;
                        DoXfer(addr,pro,lev,dst,mode,oct,M_SEND|M_STILL,buf);
                    } else {
                        Token1; Token1; Token1;
                        DoXfer(addr,pro,lev,dst,mode,oct,M_SEND,buf);
                    }
                }
            } else {
                    if ((motion || origin) && nzflag) { /* was origin && nzfla
                    Token1; Token1; Token0;
                    DoZero(addr,dst,mode,oct);
                } else {
                    Token0;
                    mode[oct]=M_STOP;
                }
            }
        }
        break;
    case M_STILL:
        GetData(addr,new,src);
        nz=Decide(new); nzflag=nz<=thresh[octs-oct];
        if (nzflag || Quantize(new,zero,pro,lev,kle->frmh.quantizer[octs-oct])
            Token0;
            mode[oct]=M_STOP;
        } else {
            Token1;
            DoXfer(addr,pro,lev,dst,mode,oct,M_STILL,buf);
```

```
            }
            break;
        case M_LPF|M_STILL:
            GetData(addr,new,src);
            QuantizeLPF(new,pro,lev,kle->frmh.quantizer[0]);
            VerifyData(lev[0],bitmask,tmp);
            VerifyData(lev[1],bitmask,tmp);
            VerifyData(lev[2],bitmask,tmp);
            VerifyData(lev[3],bitmask,tmp);
            IntEncLev(lev,kle->seqh.precision-kle->frmh.quantizer[0],buf);
            PutData(addr,pro,dst);
            mode[oct]=M_QUIT;
            break;
        case M_LPF|M_SEND:
            GetData(addr,new,src);
            GetData(addr,old,dst);
            no=DecideDelta(new,old); noflag=no<=compare[octs-oct];
            if (noflag) {
                Token0;
            } else {
                Token1;
                Quantize(new,old,pro,lev,kle->frmh.quantizer[0]);
                HuffEncLev(lev,buf);
                PutData(addr,pro,dst);
            }
            mode[oct]=M_QUIT;
            break;
        }
        switch(mode[oct]) {
        case M_STOP:
            StopCounters(mode,oct,mask,blk,x,y,octs);
            break;
        case M_QUIT:
            break;
        default:
            DownCounters(mode,oct,mask,blk);
            break;
        }
    } while (mode[oct]!=M_QUIT);
    }
}

void    KlicsEncUV(short *src,short *dst,int octs,int size[2],int thresh[5], int c
{
    int     oct, mask, x, y, X, Y, sub, tmp, step=4<<octs, blk[4], mode[4], nz, no
    int     addr[4], new[4], old[4], pro[4], lev[4], zero[4]={0,0,0,0};
    Boolean nzflag, noflag, origin;
    int     bitmask=-1<<kle->seqh.precision-kle->frmh.quantizer[0]-1;
    Buf     buf=&kle->buf;

    for(Y=0;Y<size[1];Y+=step)
    for(X=0;X<size[0];X+=step)
    for(y=Y;y<size[1] && y<Y+step;y+=step>>1)
    for(x=X;x<size[0] && x<X+step;x+=step>>1)
    for(sub=0;sub<4;sub++) {
    mode[oct=octs-1]=base_mode;
    if (sub==0) mode[oct=octs-1] |= M_LPF;
    mask=2<<oct;
    do {
        GetAddr(addr,x,y,sub,oct,size,mask);
        switch(mode[oct]) {
        case M_VOID:
            GetData(addr,old,dst);
```

Engineering:KlicsCode:CompPict:KlicsEnc.c

```
        if (BlkZero(old)) mode[oct]=M_STOP;
        else { DoZero(addr,dst,mode,oct); }
        break;
    case M_SEND|M_STILL:
        GetData(addr,new,src);
        nz=Decide(new); nzflag=nz<=thresh[octs-oct];
        if (nzflag || Quantize(new,zero,pro,lev,kle->frmh.quantizer[octs-oct])
            GetData(addr,old,dst);
            if (BlkZero(old)) {
                Token0;
                mode[oct]=M_STOP;
            } else {
                Token1; Token1;
                DoZero(addr,dst,mode,oct);
            }
        } else {
            Token1; Token0;
            DoXfer(addr,pro,lev,dst,mode,oct,M_SEND|M_STILL,buf);
        }
        break;
    case M_SEND:
        GetData(addr,new,src);
        GetData(addr,old,dst);
        nz=Decide(new); nzflag=nz<=thresh[octs-oct];
        if (BlkZero(old)) {
            if (nzflag || Quantize(new,zero,pro,lev,kle->frmh.quantizer[octs-o
                Token0;
                mode[oct]=M_STOP;
            } else {
                Token1; Token0;
                DoXfer(addr,pro,lev,dst,mode,oct,M_SEND|M_STILL,buf);
            }
        } else {
            int     oz=Decide(old), no=DecideDelta(new,old);
            Boolean motion=(nz+oz)>>oct <= no; /* motion detection */

            no=DecideDelta(new,old); noflag=no<=compare[octs-oct];
            origin=nz<=no;
                if ((!noflag || motion)  && !nzflag) { /* was !noflag && !nzfl
                if (Quantize(new,origin?zero:old,pro,lev,kle->frmh.quantizer[o
                    Token1; Token1; Token0;
                    DoZero(addr,dst,mode,oct);
                } else {
                    if (origin) {
                        Token1; Token0;
                        DoXfer(addr,pro,lev,dst,mode,oct,M_SEND|M_STILL,buf);
                    } else {
                        Token1; Token1; Token1;
                        DoXfer(addr,pro,lev,dst,mode,oct,M_SEND,buf);
                    }
                }
            } else {
                    if ((motion || origin) && nzflag) { /* was origin && nzfla
                    Token1; Token1; Token0;
                    DoZero(addr,dst,mode,oct);
                } else {
                    Token0;
                    mode[oct]=M_STOP;
                }
            }
        }
        break;
    case M_STILL:
```

Engineering:KlicsCode:CompPict:KlicsEnc.c

```
        GetData(addr,new,src);
        nz=Decide(new); nzflag=nz<=thresh[octs-oct];
        if (nzflag || Quantize(new,zero,pro,lev,kle->frmh.quantizer[octs-oct]):
            Token0;
            mode[oct]=M_STOP;
        } else {
            Token1;
            DoXfer(addr,pro,lev,dst,mode,oct,M_STILL,buf);
        }
        break;
    case M_LPF|M_STILL:
        GetData(addr,new,src);
        QuantizeLPF(new,pro,lev,kle->frmh.quantizer[0]);
        VerifyData(lev[0],bitmask,tmp);
        VerifyData(lev[1],bitmask,tmp);
        VerifyData(lev[2],bitmask,tmp);
        VerifyData(lev[3],bitmask,tmp);
        IntEncLev(lev,kle->seqh.precision-kle->frmh.quantizer[0],buf);
        PutData(addr,pro,dst);
        mode[oct]=M_QUIT;
        break;
    case M_LPF|M_SEND:
        GetData(addr,new,src);
        GetData(addr,old,dst);
        no=DecideDelta(new,old); noflag=no<=compare[octs-oct];
        if (noflag) {
            Token0;
        } else {
            Token1;
            Quantize(new,old,pro,lev,kle->frmh.quantizer[0]);
            HuffEncLev(lev,buf);
            PutData(addr,pro,dst);
        }
        mode[oct]=M_QUIT;
        break;
    }
    switch(mode[oct]) {
    case M_STOP:
        StopCounters(mode,oct,mask,blk,x,y,octs);
        break;
    case M_QUIT:
        break;
    default:
        DownCounters(mode,oct,mask,blk);
        break;
    }
  } while (mode[oct]!=M_QUIT);
  }
}

/* index to quant and vice versa */
#define i2q(i)  (float)i*HISTO_DELTA/(float)HISTO
#define q2i(q)  Fix2Long(X2Fix(q*(float)HISTO/HISTO_DELTA))


/*  Function Name:  LookAhead
 *  Description:    Examine base of tree to calculate new quantizer value
 *  Arguments:  src - source channel memory
 *              dst - destination memory (and old for videos)
 *              octs, size - octaves of decomposition and image dimensions
 *              norms - base HVS weighted normals
 *  Returns:    calculates new quant
 */
```

➡ Engineering:KlicsCode:CompPict:KlicsEnc.c

```
void    LookAhead(short *src,short *dst,float norms[5][3],KlicsE kle)
{
    int     x, y, sub, index, size[2]={kle->seqh.sequence_size[0],kle->seqh.sequen
            thresh[HISTO], quact[HISTO], target;
    int     new[4], old[4], addr[4], zero[4]={0,0,0,0};
    float   quant;

    for(index=0;index<HISTO;index++) {
        thresh[index]=0;
        quact[index]=0;
    }
    for(y=0;y<size[1];y+=2<<octs)
    for(x=0;x<size[0];x+=2<<octs)
    for(sub=1;sub<4;sub++) {
        float   q_thresh;
        int     nz, no, oz, blk;
        Boolean ozflag, origin, motion;

        GetAddr(addr,x,y,sub,octs-1,size,1<<octs);
        GetData(addr,new,src);
        GetData(addr,old,dst);
        nz=Decide(new);
        oz=Decide(old);
        no=DecideDelta(new,old);
        ozflag=kle->encd.intra || BlkZero(old);
        origin=nz<=no;
        motion=(nz+oz)>>octs <= no;
        q_thresh=(float)nz/DecideDouble(norms[1][1]);
        if (ozflag || origin) {
            float   qt=GuessQuantize(new,zero,norms[1][0]);

            q_thresh=q_thresh<qt?q_thresh:qt;
        } else {
            float   qt=GuessQuantize(new,old,norms[1][0]);

            q_thresh=q_thresh<qt?q_thresh:qt;
            if (!motion) {
                qt=(float)no/DecideDouble(norms[1][2]);
                q_thresh=q_thresh<qt?q_thresh:qt;
            }
        }
        index=q2i(q_thresh);
        index=index<0?0:index>HISTO-1?HISTO-1:index;
        thresh[index]++;
    }
    for(index=HISTO-1;index>=0;index--)
        quact[index]=thresh[index]*index+(index==HISTO-1?0:quact[index+1]);

    /* buffer must be greater than bfp_in after this frame */
    /* buffer must be less than buff_size+bfp_in */
    target=kle->encd.bpf_out*kle->encd.prevquact/kle->encd.prevbytes; /* previous
    index=1;
    while(index<HISTO && quact[index]/index>target) index++;
    quant=i2q(index);

    kle->encd.tmp_quant=(kle->encd.tmp_quant+quant)/2.0;
    kle->encd.tmp_quant=i2q((index=q2i(kle->encd.tmp_quant))); /* forward and reve

    kle->encd.prevquact=quact[index]/(index==0?1:index);
}

/*  Function Name:  BaseNormals
```

Engineering:KlicsCode:CompPict:KlicsEnc.c

```
 * Description:    Calculates base HVS weighted normals
 * Arguments:  norms - storage for normals
 * Returns:    weighted normals
 */

void    BaseNormals(float norms[5][3],KlicsE kle)
{
    float   base_norm[3]={1.0,kle->encd.thresh,kle->encd.compare};
    int     norm, oct;

    for(oct=0;oct<5;oct++)
        for(norm=0;norm<3;norm++)
            norms[oct][norm]=base_norm[norm]*kle->encd.base[oct]*(float)(1<<kl
}

/*  Function Name:  Normals
 *  Description:    Calculates HVS weighted normals @ quant
 *  Arguments:  norms - storage for normals
 *  Returns:    weighted normals and LPF bits
 */

void    Normals(float base_norms[5][3],int thresh[5],int compare[5],KlicsE kle)
{

    int     oct, i, norm;

    for(oct=0;oct<=kle->seqh.octaves[0];oct++) {
        norm=Fix2Long(X2Fix(base_norms[oct][0]*kle->encd.tmp_quant));
        norm=norm<1?1:norm;
        for(i=0;0!=(norm&-3);i++)
            norm=norm>>1;
        switch(norm) {
        case 1:
            kle->frmh.quantizer[oct]=i;
            break;
        case 2:
            kle->frmh.quantizer[oct]=i+1;
            break;
        case 3:
        case 4:
            kle->frmh.quantizer[oct]=i+2;
        }
        thresh[oct]=Fix2Long(X2Fix(DecideDouble(base_norms[oct][1]*kle->encd.tmp_q
        compare[oct]=Fix2Long(X2Fix(DecideDouble(base_norms[oct][2]*kle->encd.tmp_
    }
    kle->frmh.quantizer[0]=kle->frmh.quantizer[0]<3?3:kle->frmh.quantizer[0];
    /* minimum 4 bits of quant for lpf due to dynamic range problems */
}

Boolean KlicsFlags(KlicsE kle)
{
    Boolean skip=false;

    kle->encd.buffer-=kle->encd.bpf_in;
    kle->frmh.flags=0;
    if (kle->encd.buffer<0)
        kle->encd.buffer=0;
    if (kle->encd.intra)
        kle->frmh.flags |= KFH_INTRA;
    else
        if (skip=kle->encd.buf_sw && kle->encd.buffer>=kle->encd.buf_size)
            kle->frmh.flags |= KFH_SKIP;
    return(skip);
```

Engineering:KlicsCode:CompPict:KlicsEnc.c

```
/*
** Function Name:  KlicsEncode
 * Description:    Encode a frame from YUV (de)transformed image
 * Arguments:  src - source image(s)
 *             dst - transformed destination memory (and old for videos)
 */

long    KlicsEncode(short *src[3], short *dst[3], KlicsE kle)
{
    float   base_norms[5][3];
    int     channel, thresh[5], compare[5];
    Buf     buf=&kle->buf;

    buf_winit(buf)
    if (KlicsFlags(kle))
        kle->frmh.length=0;
    else {
        for(channel=0;channel<kle->seqh.channels;channel++) {
            int     size[2]={kle->seqh.sequence_size[0]>>(channel==0?0:kle->seqh.s
                            kle->seqh.sequence_size[1]>>(channel==0?0:kle->seqh.su
                        area=size[0]*size[1], octs=kle->seqh.octaves[channel==0?0:

            switch(kle->seqh.wavelet) {
            case WT_Haar:
                HaarForward(src[channel],size,octs);
                break;
            case WT_Daub4:
                Daub4Forward(src[channel],size,octs);
                break;
            }
        }
        BaseNormals(base_norms,kle);
        if (kle->encd.auto_q && !kle->encd.intra)
            LookAhead(src[0],dst[0],base_norms,kle);
        else
            kle->encd.tmp_quant=kle->encd.quant;
        Normals(base_norms,thresh,compare,kle);
        for(channel=0;channel<kle->seqh.channels;channel++) {
            int     size[2]={kle->seqh.sequence_size[0]>>(channel==0?0:kle->seqh.s
                            kle->seqh.sequence_size[1]>>(channel==0?0:kle->seqh.sub_sa
                        octs=kle->seqh.octaves[channel==0?0:1];

            if (kle->encd.intra)
                KLZERO(dst[channel],size[0]*size[1]);
            if (channel==0) KlicsEncY(src[channel],dst[channel],octs,size,thresh,c
            else KlicsEncUV(src[channel],dst[channel],octs,size,thresh,compare,kle
        }
        buf_flush(buf);
        kle->frmh.length=buf_size(buf);
        kle->encd.buffer+=kle->frmh.length;
        if (!kle->encd.intra)
            kle->encd.prevbytes=kle->frmh.length;
    }
    return(kle->frmh.length);
}
```

Engineering:KlicsCode:CompPict:KlicsHeader.h

```
/*******************************
 *
 * © Copyright 1993 KLICS Limited
 * All rights reserved.
 *
 * Written by: Adrian Lewis
 *
 *******************************/
/*
 * Sequence and frame headers for Klics-Encoded files
 * High byte first
 */

typedef struct {
    unsigned short  description_length; /* Fixed    - Size of this or parent struc
    unsigned char   version_number[2];  /* Fixed    - Version and revision numbers
} KlicsHeader;

typedef struct {
    KlicsHeader head;                   /* Fixed    - Size and version of this str
    unsigned short  sequence_size[3];   /* Source   - Luminance dimensions and num
    unsigned char   channels;           /* Source   - Number of channels: 3 - YUV,
    unsigned char   sub_sample[2];      /* Source   - UV sub-sampling in X and Y d
    unsigned char   wavelet;            /* Source   - Wavelet used: 0 - Haar, 1 -
    unsigned char   precision;          /* Source   - Bit precision for transform
    unsigned char   octaves[2];         /* Source   - Number of octaves Y/UV (maxi
    unsigned char   reserved[3];        /* Fixed    - Reserved for future use */
} KlicsSeqHeader;

typedef struct {
    KlicsHeader head;                   /* Fixed    - Size and version of this str
    unsigned long   length;             /* Calc     - Length of frame data (bytes)
    unsigned long   frame_number;       /* Calc     - Frame number intended for se
    unsigned char   flags;              /* Calc     - Bitfield flags: 0 - frame sk
    unsigned char   quantizer[5];       /* Calc     - Quantiser shift values[octav
    unsigned short  reserved;           /* Fixed    - Reserved for future use */
} KlicsFrameHeader;

#define KFH_SKIP    0x1
#define KFH_INTRA   0x2

/*
 * Implementation notes :
 *     QuickTime   Must have KlicsFrameHeader.length set to a valid number
 *     Sun         Must have KlicsSeqHeader in data stream
 *
 * Possible developments:
 *     KlicsFrameHeader.quantizer
 *         Currently contains shift rather than step-size
 *         Different values for UV and GH,HG,GG sub-bands are not currently suppo
 */
```

```
Engineering:KlicsCode:Klics Codec:KlicsEncode.r

/*
 *  KlicsEncode resource file
 */


#include "Types.r"
#include "MPWTypes.r"
#include "ImageCodec.r"

/*
 *  Klics Compressor included into the applications resource file here
 */

#define klicsCodecFormatName    "Klics"
#define klicsCodecFormatType    'klic'


/*

    This structure defines the capabilities of the codec. There will
    probably be a tool for creating this resource, which measures the performance
    and capabilities of your codec.

*/
resource 'cdci' (129, "Klics CodecInfo", locked) {
    klicsCodecFormatName,                          /* name of the codec TYPE ( da
    1,                                             /* version */
    1,                                             /* revision */
    'klic',                                        /* who made this codec */
    0,
    codecInfoDoes32|codecInfoDoes8|codecInfoDoesTemporal,   /* depth and etc suppo
    codecInfoDepth24|codecInfoSequenceSensitive,   /* which data formats do we un
    100,                                           /* compress accuracy (0-255) (
    100,                                           /* decompress accuracy (0-255)
    0,                                             /* millisecs to compress 320x2
    0,                                             /* millisecs to decompress 320
    0,                                             /* compression level (0-255) (
    0,
    32,                                            /* minimum height */
    32,                                            /* minimum width */
    0,
    0,
    0
};

resource 'thng' (128,  "Klics Compressor", locked) {
    compressorComponentType,
    klicsCodecFormatType,
    'klic',
    codecInfoDoes32|codecInfoDoes8|codecInfoDoesTemporal,
    0,
    'cdec',
    128,
    'STR ',
    128,
    'STR ',
    129,
    'ICON',
    128
};

resource 'STR ' (128) {
    "Klics Compress"
```

```
};

resource 'STR ' (129) {
    "Wavelet transform & multiresolution tree based coding scheme"
};
```

Engineering:KlicsCode:Klics Codec:KlicsDecode.r

```
/*
 * KlicsDecode resource file
 */


#include "Types.r"
#include "MPWTypes.r"
#include "ImageCodec.r"

/*
 * Klics Compressor included into the applications resource file here
 */

#define klicsCodecFormatName    "Klics"
#define klicsCodecFormatType    'klic'


/*

    This structure defines the capabilities of the codec. There will
    probably be a tool for creating this resource, which measures the performance
    and capabilities of your codec.

*/
resource 'cdci' (129, "Klics CodecInfo",locked) {
    klicsCodecFormatName,                        /* name of the codec TYPE ( da
    1,                                           /* version */
    1,                                           /* revision */
    'klic',                                      /* who made this codec */
    codecInfoDoes32|codecInfoDoes16|codecInfoDoes8|codecInfoDoesTemporal|codecInfo
    0,
    codecInfoDepth24|codecInfoSequenceSensitive, /* which data formats do we un
    100,                                         /* compress accuracy (0-255) (
    100,                                         /* decompress accuracy (0-255)
    0,                                           /* millisecs to compress 320x2
    0,                                           /* millisecs to decompress 320
    0,                                           /* compression level (0-255) (
    0,
    32,                                          /* minimum height */
    32,                                          /* minimum width */
    0,
    0,
    0
};

resource 'thng' (130, "Klics Decompressor",locked) {
    decompressorComponentType,
    klicsCodecFormatType,
    'klic',
    codecInfoDoes32|codecInfoDoes16|codecInfoDoes8|codecInfoDoesTemporal|codecInfo
    0,
    'cdec',
    128,
    'STR ',
    130,
    'STR ',
    131,
    'ICON',
    130
};


resource 'STR ' (130) {
```

CLAIMS

WE CLAIM:

1. A method of transforming a sequence of input digital data values into a first sequence of transformed digital data values and of inverse transforming a second sequence of transformed digital data values into a sequence of output digital data values, said sequence of input digital data values comprising a boundary subsequence and a non-boundary subsequence, comprising the steps of:

running a number of said input digital data values of said boundary subsequence through a low pass boundary forward transform perfect reconstruction digital filter and through a high pass boundary forward transform perfect reconstruction digital filter to produce a first subsequence of said first sequence of transformed digital data values, said first subsequence of said first sequence of transformed digital data values comprising interleaved low and high frequency transformed digital data values;

running a number of said input digital data values of said non-boundary subsequence through a low pass non-boundary forward transform perfect reconstruction digital filter and also through a high pass non-boundary forward transform perfect reconstruction digital filter to produce a second subsequence of said first sequence of transformed digital data values, said second subsequence of said first sequence of transformed digital data values comprising interleaved low and high frequency transformed digital data values, said low pass boundary forward transform perfect reconstruction digital filter having a fewer number of coefficients than said low pass non-boundary forward transform perfect reconstruction digital filter, said high pass boundary forward transform perfect reconstruction digital filter having a fewer number of coefficients

than said high pass non-boundary forward transform perfect reconstruction digital filter;

converting said first sequence of transformed digital data values into said second sequence of transformed digital data values, said second sequence of transformed digital data values comprising a first subsequence of said second sequence of transformed digital data values and a second subsequence of said second sequence of transformed digital data values;

running a number of said first subsequence of said second sequence of transformed digital data values through an interleaved boundary inverse transform perfect reconstruction digital filter to produce at least one output digital data value;

running a number of said second subsequence of said second sequence of transformed digital data values through a first interleaved non-boundary inverse transform perfect reconstruction digital filter to produce output digital data values; and

running a number of said second subsequence of transformed digital data values through a second interleaved non-boundary inverse transform perfect reconstruction digital filter to produce output digital data values, said output digital data values produced by said interleaved boundary inverse transform perfect reconstruction digital filter, said first interleaved non-boundary inverse transform perfect reconstruction digital filter, and said second interleaved non-boundary inverse transform perfect reconstruction digital filter comprising a subsequence of said output digital data values of said sequence of output digital data values.

2. The method of Claim 1, wherein said low pass boundary forward transform perfect reconstruction digital filter has X coefficients and wherein said low pass non-boundary forward transform perfect reconstruction digital

filter has Y coefficients, Y being greater than X, said X coefficients of said low pass boundary forward transform perfect reconstruction digital filter being chosen so that said low pass boundary forward transform perfect reconstruction digital filter outputs a transformed digital data value H_0 when the low pass boundary forward perfect transform reconstruction digital filter operates on input digital data values ID_0-ID_{X-1} adjacent said boundary, said transformed digital data value H_0 being substantially equal to what the output of the low pass non-boundary forward transform perfect reconstruction digital filter would be were the low pass non-boundary forward perfect reconstruction digital filter to operate on ID_0-ID_{X-1} as well as Y-X additional input digital data values outside said boundary, said additional input digital data values having preselected values.

3. The method of Claim 2, wherein Y-X=1, wherein there is one additional input digital data value ID_{-1}, and wherein ID_{-1} is preselected to be substantially equal to ID_0.

4. The method of Claim 2, wherein Y-X=1, wherein there is one additional input digital data value ID_{-1}, and wherein ID_{-1} is preselected to be substantially equal to zero.

5. The method of Claim 1, wherein said sequence of input digital data values is a sequence of digital data values associated with pixels of either a row or a column of a two dimensional image, said boundary of said sequence of input digital data values corresponding with either a start or an end of said row or said column.

6. The method of Claim 1, wherein said sequence of input digital data values is a sequence of digital data values associated with an audio signal.

7. The method of Claim 1, wherein said low and high pass non-boundary forward transform perfect reconstruction digital filters are forward transform quasi-perfect reconstruction filters which have coefficients which approximate the coefficients of true forward transform perfect reconstruction filters.

8. The method of Claim 1, wherein said low and high pass non-boundary forward transform perfect reconstruction digital filters are both four coefficient quasi-Daubechies filters the coefficients of which approximate the coefficients of true four coefficient Daubechies filters.

9. The method of Claim 8, wherein one of said four coefficient quasi-Daubechies filters has the coefficients 11/32, 19/32, 5/32 and 3/32 independent of sign.

10. The method of Claim 1, wherein said low pass non-boundary forward transform perfect reconstruction digital filter is a four coefficient quasi-Daubechies filter H of the form:

$$H_n = aID_{2n-1} + bID_{2n} + cID_{2n+1} - dID_{2n+2}$$

n being a positive integer, ID_0-ID_m being input digital data values, m being a positive integer, ID_0 being the first input digital data value in said sequence of input digital data values, and wherein said low pass boundary forward transform perfect reconstruction digital filter is a three coefficient digital filter of the form:

$$H_0 = aID_{-1} + bID_0 + cID_1 - dID_2$$

ID_{-1} being a predetermined input digital data value outside said boundary and having a preselected value.

11. The method of Claim 10, wherein said high pass

non-boundary forward transform perfect reconstruction digital filter is a four coefficient quasi-Daubechies filter of the form:

$$G_n = dID_{2n-1} + cID_{2n} - bID_{2n+1} + aID_{2n+2}$$

n being a positive integer, and wherein said high pass boundary forward transform perfect reconstruction digital filter is a three coefficient digital filter of the form:

$$G_0 = dID_{-1} + cID_0 - bID_1 + aID_2$$

dID_{-1} having a preselected value.

12. The method of Claim 11, wherein: a + b + c - d is substantially equal to 1, wherein a - b + c + d is substantially equal to 0, and wherein ac - bd is substantially equal to zero.

13. The method of Claim 12, wherein: a=11/32, b=19/32, c=5/32 and d=3/32.

14. The method of Claim 11, wherein said interleaved boundary inverse transform perfect reconstruction digital filter is a two coefficient digital filter of the form:

$$OD_0 = 4(b-a)H_0 + 4(c-d)G_0$$

wherein OD_0 is an output digital data value of said sequence of output digital data values, wherein G_0 is the output of said high pass boundary forward transform perfect reconstruction digital filter when the high pass boundary forward transform perfect reconstruction digital filter operates on input digital data values ID_0, ID_1 and ID_2 adjacent said boundary, and wherein H_0 is the output of said low pass boundary forward transform perfect reconstruction digital filter when the low pass boundary

forward transform perfect reconstruction digital filter operates on input digital data values ID_0, ID_1 and ID_2 adjacent said boundary.

15. The method of Claim 14, wherein one of said first and second interleaved non-boundary inverse transform perfect reconstruction digital filters is of the form:

$$D_{2n+1} = 2(cH_n - bG_n + aH_{n+1} + dG_{n+1})$$

n being a non-negative integer, and wherein the other of said first and second interleaved non-boundary inverse perfect reconstruction digital filters is of the form:

$$D_{2n+2} = 2(-dH_n + aG_n + bH_{n+1} + cG_{n+1})$$

n being a non-negative integer, wherein H_n, G_n, H_{n+1} and G_{n+1} comprise a subsequence of said second sequence of transformed digital data values.

16. The method of Claim 1, wherein said low pass non-boundary forward transform perfect reconstruction digital filter is a four coefficient quasi-Daubechies filter having the coefficients: 11/32, 19/32, 5/32 and -3/32, and wherein said high pass non-boundary forward transform perfect reconstruction digital filter is a four coefficient quasi-Daubechies filter having the coefficients: 3/32, 5/32, -19/32 and 11/32.

17. The method of Claim 1, wherein said low and high pass non-boundary forward transform perfect reconstruction digital filters are chosen from the group consisting of: true six coefficient Daubechies filters and quasi-Daubechies filters, the coefficients of the quasi-Daubechies filters approximating the coefficients of true six coefficient Daubechies filters.

18. The method of Claim 1, further comprising the steps of:

encoding said first sequence of transformed digital data values into an encoded sequence; and

decoding said encoded sequence of digital data values into said second sequence of transformed digital data values and supplying said second sequence of transformed digital data values to said interleaved boundary inverse transform perfect reconstruction digital filter, said first interleaved non-boundary inverse transform perfect reconstruction digital filter, and said second interleaved non-boundary inverse transform perfect reconstruction digital filter.

19. The method of Claim 18, further comprising the step of:

quantizing each of said digital data values in said first sequence of transformed values before said encoding step.

20. The method of Claim 1, wherein each of said input digital data values of said sequence of input digital data values is stored in a separate memory location, and wherein some of said memory locations are overwritten in a sequence with said sequence of transformed digital data values as said digital data input values are transformed into said transformed digital data values.

21. A method of transforming a sequence of input digital data values into a sequence of transformed digital data values, said sequence of input digital data values comprising a boundary subsequence and a non-boundary subsequence, comprising the steps of:

running a number of said input digital data values of said boundary subsequence through a low pass boundary forward transform perfect reconstruction

digital filter and through a high pass boundary forward transform perfect reconstruction digital filter to produce a first subsequence of said sequence of transformed digital data values, said first subsequence of said sequence of transformed digital data values comprising interleaved low and high frequency transformed digital data values; and

running a number of said input digital data values of said non-boundary subsequence through a low pass non-boundary forward transform perfect reconstruction digital filter and also through a high pass non-boundary forward transform perfect reconstruction digital filter to produce a second subsequence of said sequence of transformed digital data values, said second subsequence of said sequence of transformed digital data values comprising interleaved low and high frequency transformed digital data values, said low pass boundary forward transform perfect reconstruction digital filter having a fewer number of coefficients than said low pass non-boundary forward transform perfect reconstruction digital filter, said high pass boundary forward transform perfect reconstruction digital filter having a fewer number of coefficients than said high pass non-boundary forward transform perfect reconstruction digital filter.

22. A method, comprising the steps of:

generating a sub-band decomposition having a plurality of octaves, a first of said plurality of octaves comprising at least one first digital data value, a second of said plurality of octaves comprising at least one second digital data value;

calculating a sum of the absolute values of said at least one first digital data value;

determining if said at least one first digital data value is interesting using a first threshold

limit;

calculating a sum of the absolute values of said at least one second digital data value; and

determining if said at least one second digital data value is interesting using a second threshold limit.

23. A method of traversing a tree decomposition, said tree decomposition comprising a plurality of transformed data values, each of said plurality of transformed data values having a unique address identified by coordinates X and Y, comprising the step of:

calculating at least four transformed data value addresses by incrementing a count, the count comprising one bit $C1_x$ in the X coordinate and one bit $C1_y$ in the Y coordinate, to generate said at least four transformed data value addresses.

24. A method, comprising the step of:

determining an address of a transformed data value in a tree decomposition by shifting a value a number of times, said tree decomposition having a number of octaves, said transformed data value being in one of said octaves, said number of times being at least dependent upon said one octave.

25. A method, comprising the step of:

determining an address of a transformed data value in a tree decomposition by multiplying a value by a factor, said tree decomposition having a number of octaves, said transformed data value being in one of said octaves, said factor being at least dependent upon said one octave.

26. A method, comprising the step of:

determining an address of a transformed data value in a tree decomposition by shifting a value a number of times, said tree decomposition having a number of frequency sub-

bands, said transformed data value being in one of said frequency sub-bands, said number of times being at least dependent upon said frequency sub-band.

27. A method, comprising the step of:

determining an address of a transformed data value in a tree decomposition by performing a logical operation upon a value, said tree decomposition having a number of frequency sub-bands, said transformed data value being in one of said frequency sub-bands, said logical operation performed being at least dependent upon said one frequency sub-band.

28. The method of Claim 27, wherein said logical operation is a bit-wise logical AND operation.

29. A method for determining a low pass quasi-perfect reconstruction filter and a high pass quasi-perfect reconstruction filter from a wavelet function, said low pass quasi-perfect reconstruction filter having a plurality of coefficients, said high pass quasi-perfect reconstruction filter having a plurality of coefficients, comprising the steps of:

determining a low pass wavelet digital filter and a high pass wavelet digital filter from said wavelet function, said low pass wavelet digital filter having a plurality of coefficients, said high pass wavelet digital filter having a plurality of coefficients;

choosing the coefficients of said low pass quasi-perfect reconstruction digital filter to be fractions such that when a sequence of data values having values of 1 is processed by said low pass quasi-perfect reconstruction digital filter the output of said low pass quasi-perfect reconstruction digital filter is exactly a power of 2; and

choosing the coefficients of the high pass quasi-perfect reconstruction digital filter to be fractions such that when a sequence of data values having values of 1 is

processed by said high pass quasi-perfect reconstruction digital filter the output of said high pass quasi-perfect reconstruction digital filter is exactly 0, whereby each of the plurality of coefficients of said low pass quasi-perfect reconstruction digital filter is substantially identical to a corresponding one of said plurality of coefficients of said low pass wavelet digital filter, and whereby each of the plurality of coefficients of said high pass quasi-perfect reconstruction digital filter is substantially identical to a corresponding one of said plurality of coefficients of said high pass wavelet digital filter.

30. A method of estimating a compression ratio of a number of original data values to a number of compressed data values at a value of a quality factor Q, comprising the steps of:

examining a first block of transformed data values of a tree, said first block being one of a number of lowest frequency blocks of a high pass component sub-band, said tree being part of a sub-band decomposition; and

determining a value of said quality factor Q at which said data values of said first block would be converted into compressed data values, and not determining a value of said quality factor Q at which any other block of data values of said tree would be converted into a number of compressed data values.

31. The method of Claim 30, wherein said number of original data values represents a frame of an image.

32. The method of Claim 31, further comprising the step of:

determining a number of lowest frequency blocks of said high pass component sub-band which would be converted into compressed data values given a value of said quality factor Q.

33. A method of transforming a sequence of image data values, comprising the step of:

filtering said sequence of image data values using a quasi-perfect reconstruction filter to generate a decomposition having a plurality of octaves, said quasi-perfect reconstruction filter having six coefficients.

34. The method of Claim 33, wherein said six coefficients are selected from the group consisting of: 30/128, 73/128, 41/128, 12/128, 7/128 and 3/128, irrespective of sign.

35. A method of detecting motion in a tree decomposition, said tree decomposition comprising a plurality of octaves of blocks of data values, comprising the steps of:

comparing data values of a first block in an octave with data values of a second block in said octave; and

generating a token indicating motion based on said comparing.

36. A method, comprising the steps of:

generating a sub-band decomposition having a plurality of octaves, a first of said plurality of octaves comprising at least one first digital data value, a second of said plurality of octaves comprising at least one second digital data value;

determining if said at least one first digital data value is interesting using a first threshold limit; and

determining if said at least one second digital data value is interesting using a second threshold limit.

37. A method, comprising the steps of:

generating a sub-band decomposition of a first frame having a plurality of octaves, a first of said plurality of octaves comprising at least one first digital data value, a

second of said plurality of octaves comprising at least one second digital data value;

generating a sub-band decomposition of a second frame having a plurality of octaves, a first of said plurality of octaves comprising at least one first digital data value, a second of said plurality of octaves comprising at least one second digital data value;

comparing said first digital data value of said first frame with said first digital data value of said second frame using a first threshold compare; and

comparing said second digital data value of said first frame with said second digital data value of said second frame using a second threshold compare.

38. A method, comprising the steps of:

reading a sequence of data values from a plurality of memory locations, each of said data values being stored in a separate one of said plurality of memory locations; and

overwriting some of said memory locations in a sequence as said data values are transformed into a sequence of transformed data values of a sub-band decomposition.

39. A method, comprising the steps of:

performing a function on a plurality of data values of a new block to generate a first output value, said new block being a block of data values of a sub-band decomposition of a new frame;

performing said function on a plurality of numbers to generate a second output value, each of said numbers substantially equalling a difference of a data value in said plurality of data values of said new block and a corresponding data value in a corresponding plurality of data values of an old block, said old block being a block of data values of a sub-band decomposition of an old frame; and

generating a token if said first output value has a

predetermined relationship with respect to said second output value.

40. The method of Claim 39, wherein said token is a SEND_STILL token.

41. A method, comprising the steps of:

performing a function on a plurality of data values of a new block to generate a corresponding plurality of output values, said new block being a block of data values of a sub-band decomposition;

comparing each of said plurality of output values with a predetermined number; and

generating a token if substantially all of said output values have a predetermined relationship with respect to said predetermined number.

42. The methcd of Claim 41, wherein said token is a VOID token.

43. A method, comprising the steps of:

subtracting each one of a plurality of data values of a new block with a corresponding one of a plurality of data values of a old block to generate a corresponding plurality of output values, said new block being a block of data values of a sub-band decomposition of a new frame, said old block being a block of data values of a sub-band decomposition of a old frame;

comparing each of said plurality of output values with a predetermined number; and

generating a token if substantially all of said output values have a predetermined relationship with respect to said predetermined number.

44. The method of Claim 43, wherein said token is a VOID token.

45. A method, comprising the steps of:

determining an absolute value for each of a plurality of data values of a block of a sub-band decomposition;

determining a sum of said absolute values; and

generating a token based on a comparison of said sum with a predetermined number.

46. The method of Claim 45, wherein said token is a VOID token.

47. A method, comprising the steps of:

processing a sequence of first image data values using a low pass forward transform perfect reconstruction digital filter and a high pass forward transform perfect reconstruction digital filter to create a first sequence of transformed data values, said low pass forward transform perfect reconstruction digital filter and said high pass forward transform perfect reconstruction digital filter each having coefficients chosen from a first group of coefficients independent of sign;

converting said first sequence of transformed data values into a second sequence of transformed data values; and

using digital circuitry to process said second sequence of transformed data values using a low pass inverse transform perfect reconstruction digital filter and a high pass inverse transform perfect reconstruction digital filter into a sequence of second image data values, said low pass inverse transform perfect reconstruction digital filter and said high pass inverse transform perfect reconstruction digital filter each having coefficients chosen from a second group of coefficients independent of sign.

48. The method of claim 47, wherein said digital circuitry used to process said second sequence of transformed data values is a digital computer having a

microprocessor.

49. The method of claim 47, wherein at least one of the coefficients in said first group of coefficients is not contained in said second group of coefficients.

50. The method of claim 47, wherein said first group of coefficients has a different number of coefficients than said second group of coefficients.

51. The method of claim 50, wherein said sequence of first image data values is a sequence of chrominance data values.

52. The method of claim 50, wherein said low pass forward transform perfect reconstruction digital filter and said high pass forward transform perfect reconstruction digital filter each have four coefficients, and wherein said low pass inverse transform perfect reconstruction digital filter and said high pass inverse transform perfect reconstruction digital filter each have two coefficients.

53. The method of claim 52, wherein said sequence of first image data values is a sequence of chrominance data values.

54. The method of claim 47, wherein each of said coefficients of said low pass inverse transform perfect reconstruction digital filter and said high pass inverse transform perfect reconstruction digital filter is selected from the group consisting of: 5/8, 3/8 and 1/8, independent of sign.

55. The method of claim 47, wherein said converting step comprises the steps of:

encoding said first sequence of transformed data values into a compressed data stream; and

decoding said compressed data stream into said second sequence of transformed data values.

56. A method comprising the step of using digital circuitry to process a sequence of image data values using a low pass forward transform perfect reconstruction digital filter and a high pass forward transform perfect reconstruction digital filter to generate a sub-band decomposition, said low pass forward transform perfect reconstruction digital filter and said high pass forward transform perfect reconstruction digital filter each having four coefficients, each of said four coefficients being selected from the group consisting of: 5/8, 3/8 and 1/8, independent of sign.

57. The method of claim 56, wherein said digital circuitry comprises means for low pass forward transform perfect reconstruction digital filtering and for high pass forward transform perfect reconstruction digital filtering.

58. A method comprising the step of using digital circuitry to process a sequence of transformed data values of a sub-band decomposition using an odd inverse transform perfect reconstruction digital filter and an even inverse transform perfect reconstruction digital filter, said odd inverse transform perfect reconstruction digital filter and said even inverse transform perfect reconstruction digital filter each having four coefficients, each of said four coefficients being selected from the group consisting of: 5/8, 3/8 and 1/8, independent of sign.

59. The method of claim 58, wherein said digital circuitry is a digital computer having a microprocessor.

60. A method comprising the step of generating a compressed data stream indicative of a video sequence from a sub-band decomposition, said compressed data stream

comprising a first data value, a first token, a second data value, and a second token, said first token being indicative of a first encoding method used to encode said first data value, said second token being indicative of a second encoding method used to encode said second data value, said first token consisting of a first number of bits and said second token consisting of a second number of bits.

61. The method of claim 60, wherein said first encoding method is taken from the group consisting of: SEND mode, STILL_SEND mode, VOID mode, and STOP mode.

62. The method of claim 60, wherein said first token is a single bit token.

63. A method, comprising the steps of:

forward transforming image data values to generate a first sequence of transformed data values of a first sub-band decomposition, said first sub-band decomposing having a first number of octaves;

converting said first sequence of transformed data values into a second sequence of transformed data values;

using digital circuitry to inverse transforming said second sequence of transformed data values into a third sequence of transformed data values, said third sequence of transformed data values comprising a second sub-band decomposition having a second number of octaves, said second number of octaves being smaller than said first number of octaves, said second sub-band decomposition having a low pass component, said low pass component of said second sub-band decomposition comprising data values indicative of rows of data values of an image, said rows of said image extending in a first dimension, said image also having columns of said data values extending in a second dimension;

expanding said low pass component in said first

dimension using interpolation to generate an interpolated low pass component; and

expanding said interpolated low pass component in said second dimension by replicating rows of said data values of said interpolated low pass component.

64. The method of claim 63, wherein said digital circuitry is a digital computer having a microprocessor.

65. The method of claim 63, wherein said converting step comprises the steps of:

encoding said first sequence of transformed data values into a compressed data stream comprising tokens and encoded data values; and

decoding said compressed data stream into said second sequence of transformed data values.

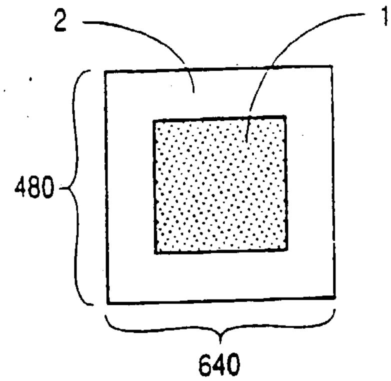


Fig. 1
(PRIOR ART)

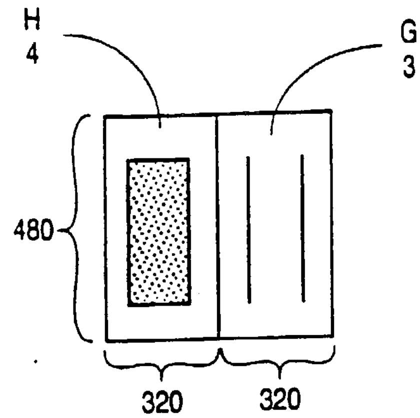


Fig. 2
(PRIOR ART)

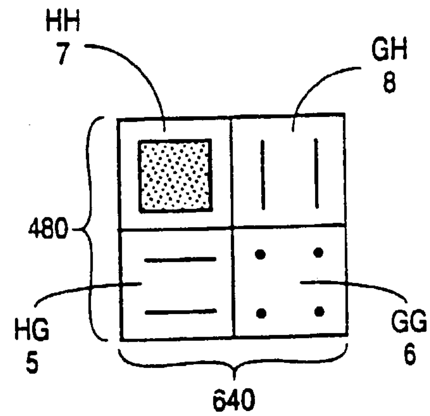


Fig. 3
(PRIOR ART)

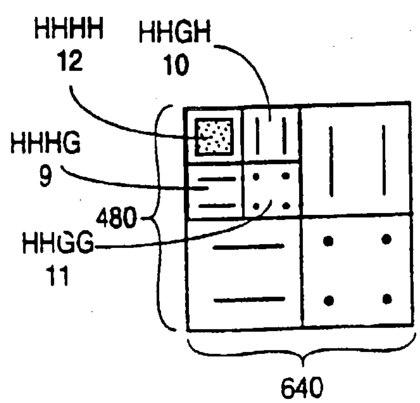


Fig. 4
(PRIOR ART)

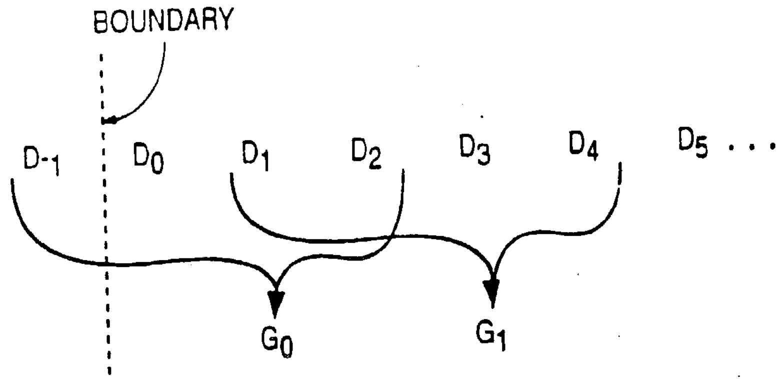


Fig. 5
(PRIOR ART)

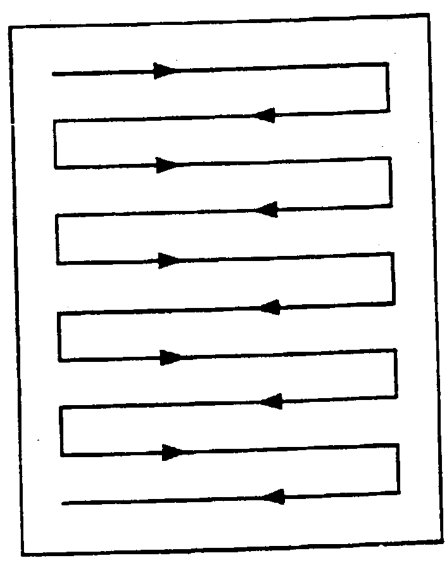


Fig. 6
(PRIOR ART)

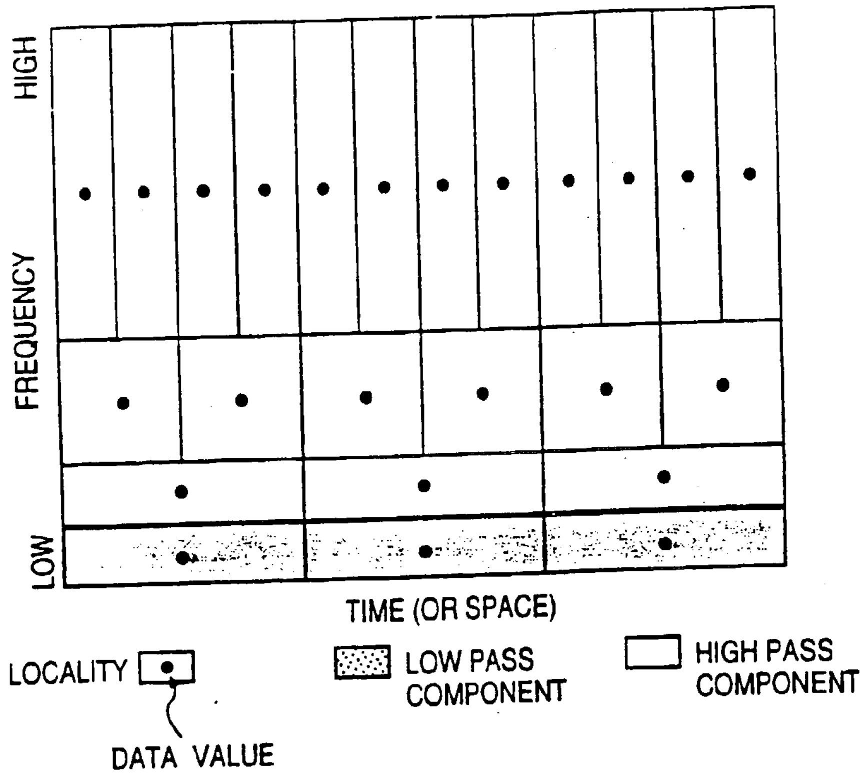


Fig. 7

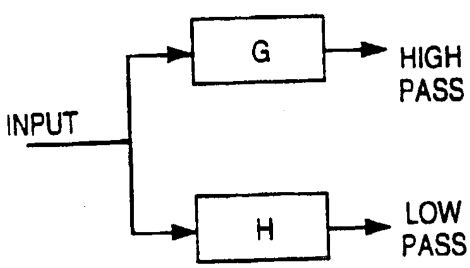


Fig. 8

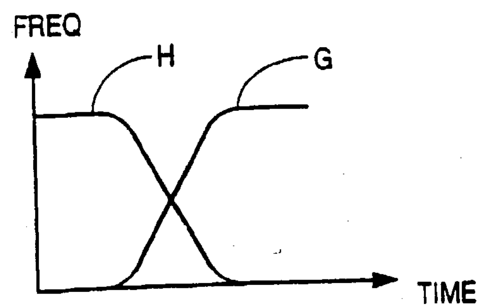


Fig. 9

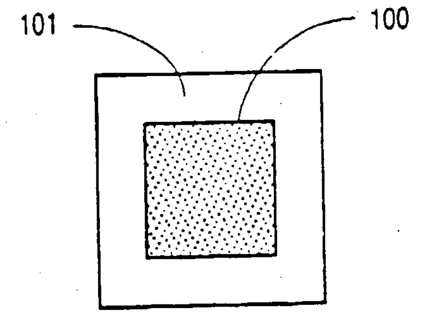


Fig. 10

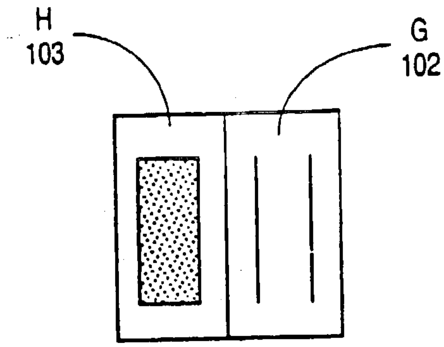


Fig. 11

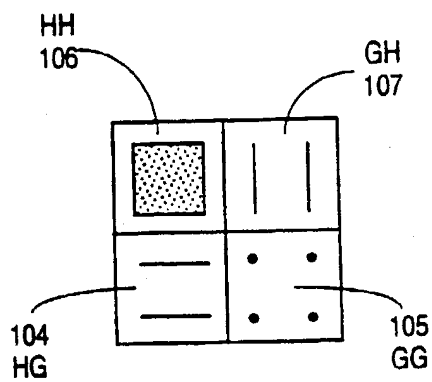


Fig. 14

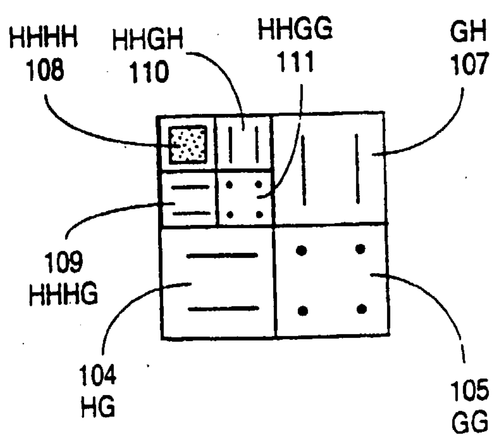


Fig. 15

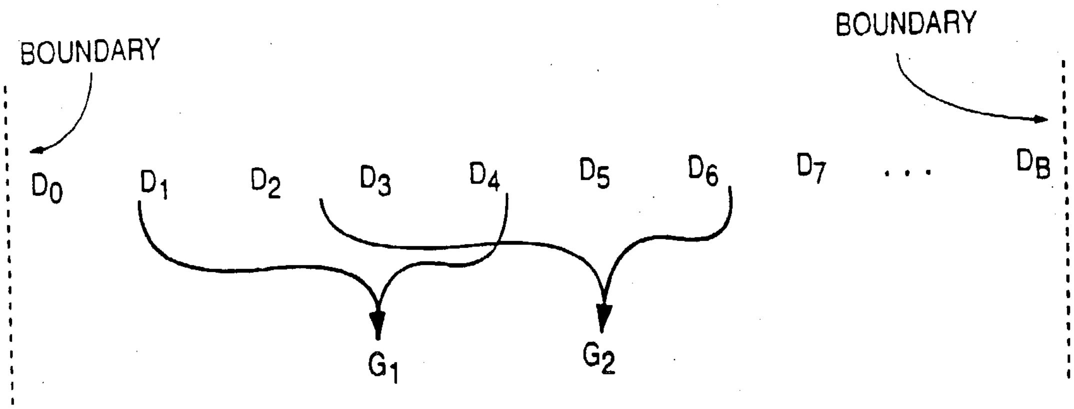

BOUNDARY
BOUNDARY
D_0
D_1
D_2
D_3
D_4
D_5
D_6
D_7
. . .
D_B
G_1
G_2

Fig. 12

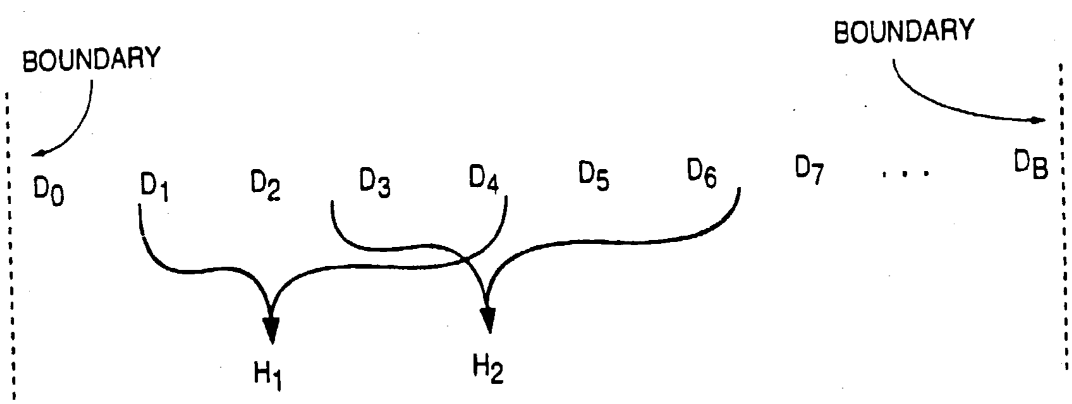

BOUNDARY
BOUNDARY
D_0
D_1
D_2
D_3
D_4
D_5
D_6
D_7
. . .
D_B
H_1
H_2

Fig. 13

COLUMN

ROW	0	1	2	3	4	5	6	7	8	9	A	B
0	D_{00}	D_{01}	D_{02}	D_{03}	D_{04}	D_{05}	D_{06}	D_{07}	D_{08}	D_{09}	D_{0A}	D_{0B}
1	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}	D_{16}	D_{17}	D_{18}	D_{19}	D_{1A}	D_{1B}
2	D_{20}	D_{21}	D_{22}	D_{23}	D_{24}	D_{25}	D_{26}	D_{27}	D_{28}	D_{29}	D_{2A}	D_{2B}
3	D_{30}	D_{31}	D_{32}	D_{33}	D_{34}	D_{35}	D_{36}	D_{37}	D_{38}	D_{39}	D_{3A}	D_{3B}
4	D_{40}	D_{41}	D_{42}	D_{43}	D_{44}	D_{45}	D_{46}	D_{47}	D_{48}	D_{49}	D_{4A}	D_{4B}
5	D_{50}	D_{51}	D_{52}	D_{53}	D_{54}	D_{55}	D_{56}	D_{57}	D_{58}	D_{59}	D_{5A}	D_{5B}
6	D_{60}	D_{61}	D_{63}	D_{63}	D_{64}	D_{65}	D_{66}	D_{67}	D_{68}	D_{69}	D_{6A}	D_{6B}
7	D_{70}	D_{71}	D_{72}	D_{73}	D_{74}	D_{75}	D_{76}	D_{77}	D_{78}	D_{79}	D_{7A}	D_{7B}
8	D_{80}	D_{81}	D_{82}	D_{83}	D_{84}	D_{85}	D_{86}	D_{87}	D_{88}	D_{89}	D_{8A}	D_{8B}
9	D_{90}	D_{91}	D_{92}	D_{93}	D_{94}	D_{95}	D_{96}	D_{97}	D_{98}	D_{99}	D_{9A}	D_{9B}
A	D_{A0}	D_{A1}	D_{A2}	D_{A3}	D_{A4}	D_{A5}	D_{A6}	D_{A7}	D_{A8}	D_{A9}	D_{AA}	D_{AB}
B	D_{B0}	D_{B1}	D_{B2}	D_{B3}	D_{B4}	D_{B5}	D_{B6}	D_{B7}	D_{B8}	D_{B9}	D_{BA}	D_{BB}

Fig. 16

ROW \ COLUMN	0	1	2	3	4	5	6	7	8	9	A	B
0	HH_{00}	GH_{00}	HH_{01}	GH_{01}	HH_{02}	GH_{02}	HH_{03}	GH_{03}	HH_{04}	GH_{04}	HH_{05}	GH_{05}
1	HG_{00}	GG_{00}	HG_{01}	GG_{01}	HG_{02}	GG_{02}	HG_{03}	GG_{03}	HG_{04}	GG_{04}	HG_{05}	GG_{05}
2	HH_{10}	GH_{10}	HH_{11}	GH_{11}	HH_{12}	GH_{12}	HH_{13}	GH_{13}	HH_{14}	GH_{14}	HH_{15}	GH_{15}
3	HG_{10}	GG_{10}	HG_{11}	GG_{11}	HG_{12}	GG_{12}	HG_{13}	GG_{13}	HG_{14}	GG_{14}	HG_{15}	GG_{15}
4	HH_{20}	GH_{20}	HH_{21}	GH_{21}	HH_{22}	GH_{22}	HH_{23}	GH_{23}	HH_{24}	GH_{24}	HH_{25}	GH_{25}
5	HG_{20}	GG_{20}	HG_{21}	GG_{21}	HG_{22}	GG_{22}	HG_{23}	GG_{23}	HG_{24}	GG_{24}	HG_{25}	GG_{25}
6	HH_{30}	GH_{30}	HH_{31}	GH_{31}	HH_{32}	GH_{32}	HH_{33}	GH_{33}	HH_{34}	GH_{34}	HH_{35}	GH_{35}
7	HG_{30}	GG_{30}	HG_{31}	GG_{31}	HG_{32}	GG_{32}	HG_{33}	GG_{33}	HG_{34}	GG_{34}	HG_{35}	GG_{35}
8	HH_{40}	GH_{40}	HH_{41}	GH_{41}	HH_{42}	GH_{42}	HH_{43}	GH_{43}	HH_{44}	GH_{44}	HH_{45}	GH_{45}
9	HG_{40}	GG_{40}	HG_{41}	GG_{41}	HG_{42}	GG_{42}	HG_{43}	GG_{43}	HG_{44}	GG_{44}	HG_{45}	GG_{45}
A	HH_{50}	GH_{50}	HH_{51}	GH_{51}	HH_{52}	GH_{52}	HH_{53}	GH_{53}	HH_{54}	GH_{54}	HH_{55}	GH_{55}
B	HG_{50}	GG_{50}	HG_{51}	GG_{51}	HG_{52}	GG_{52}	HG_{53}	GG_{53}	HG_{54}	GG_{54}	HG_{55}	GG_{55}

Fig. 17

COLUMN

ROW	0	1	2	3	4	5	6	7	8	9	A	B
0	$HHHH_{00}$	GH_{00}	$HHGH_{00}$	GH_{01}	$HHHH_{01}$	GH_{02}	$HHGH_{01}$	GH_{03}	$HHHH_{02}$	GH_{04}	$HHGH_{02}$	GH_{05}
1	HG_{00}	GG_{00}	HG_{01}	GG_{01}	HG_{02}	GG_{02}	HG_{03}	GG_{03}	HG_{04}	GG_{04}	HG_{05}	GG_{05}
2	$HHHG_{00}$	GH_{10}	$HHGG_{00}$	GH_{11}	$HHHG_{01}$	GH_{12}	$HHGG_{01}$	GH_{13}	$HHHG_{02}$	GH_{14}	$HHGG_{02}$	GH_{15}
3	HG_{10}	GG_{10}	HG_{11}	GG_{11}	HG_{12}	GG_{12}	HG_{13}	GG_{13}	HG_{14}	GG_{14}	HG_{15}	GG_{15}
4	$HHHH_{10}$	GH_{20}	$HHGH_{10}$	GH_{21}	$HHHH_{11}$	GH_{22}	$HHGH_{11}$	GH_{23}	$HHHH_{12}$	GH_{24}	$HHGH_{12}$	GH_{25}
5	HG_{20}	GG_{20}	HG_{21}	GG_{21}	HG_{22}	GG_{22}	HG_{23}	GG_{23}	HG_{24}	GG_{24}	HG_{25}	GG_{25}
6	$HHHG_{10}$	GH_{30}	$HHGG_{10}$	GH_{31}	$HHHG_{11}$	GH_{32}	$HHGG_{11}$	GH_{33}	$HHHG_{12}$	GH_{34}	$HHGG_{12}$	GH_{35}
7	HG_{30}	GG_{30}	HG_{31}	GG_{31}	HG_{32}	GG_{32}	HG_{33}	GG_{33}	HG_{34}	GG_{34}	HG_{35}	GG_{35}
8	$HHHH_{20}$	GH_{40}	$HHGH_{20}$	GH_{41}	$HHHH_{21}$	GH_{42}	$HHGH_{21}$	GH_{43}	$HHHH_{22}$	GH_{44}	$HHGH_{22}$	GH_{45}
9	HG_{40}	GG_{40}	HG_{41}	GG_{41}	HG_{42}	GG_{42}	HG_{43}	GG_{43}	HG_{44}	GG_{44}	HG_{45}	GG_{45}
A	$HHHG_{20}$	GH_{50}	$HHGG_{20}$	GH_{51}	$HHHG_{21}$	GH_{52}	$HHGG_{21}$	GH_{53}	$HHHG_{22}$	GH_{54}	$HHGG_{22}$	GH_{55}
B	HG_{50}	GG_{50}	HG_{51}	GG_{51}	HG_{52}	GG_{52}	HG_{53}	GG_{53}	HG_{54}	GG_{54}	HG_{55}	GG_{55}

Fig. 18

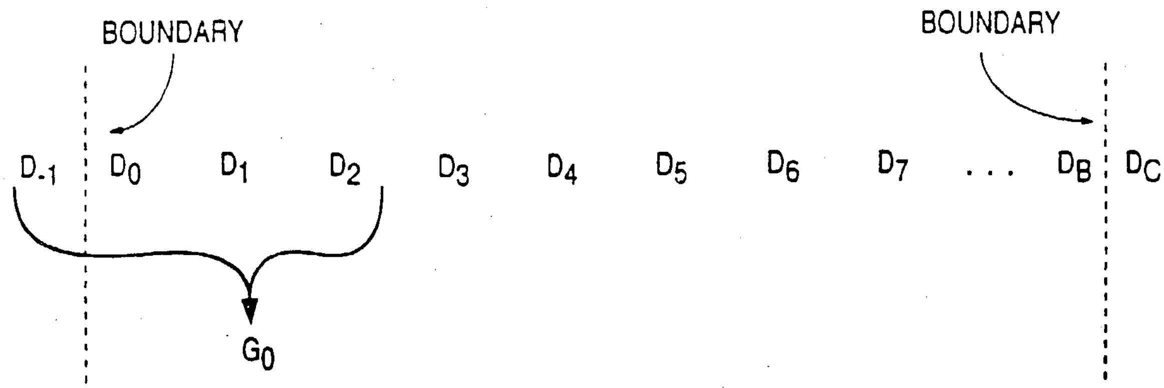

BOUNDARY
BOUNDARY
D_{-1} D_0 D_1 D_2 D_3 D_4 D_5 D_6 D_7 . . . D_B D_C
G_0

Fig. 19

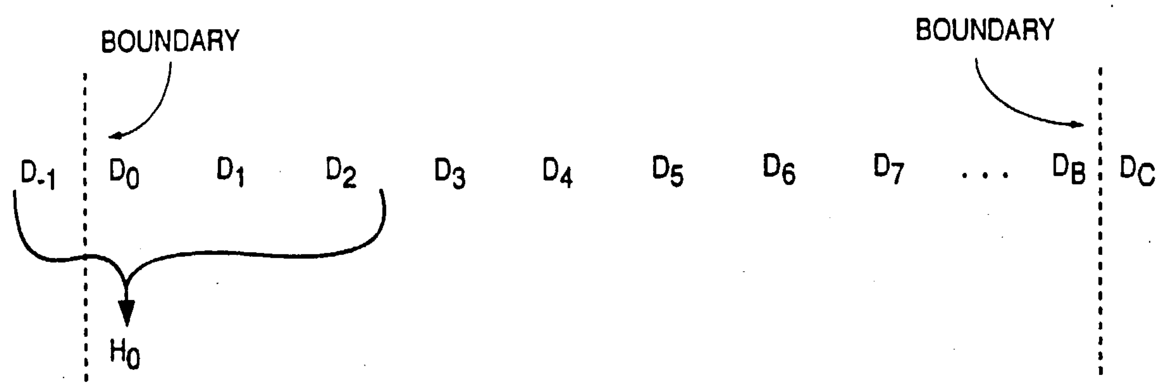

BOUNDARY
BOUNDARY
D_{-1} D_0 D_1 D_2 D_3 D_4 D_5 D_6 D_7 . . . D_B D_C
H_0

Fig. 20

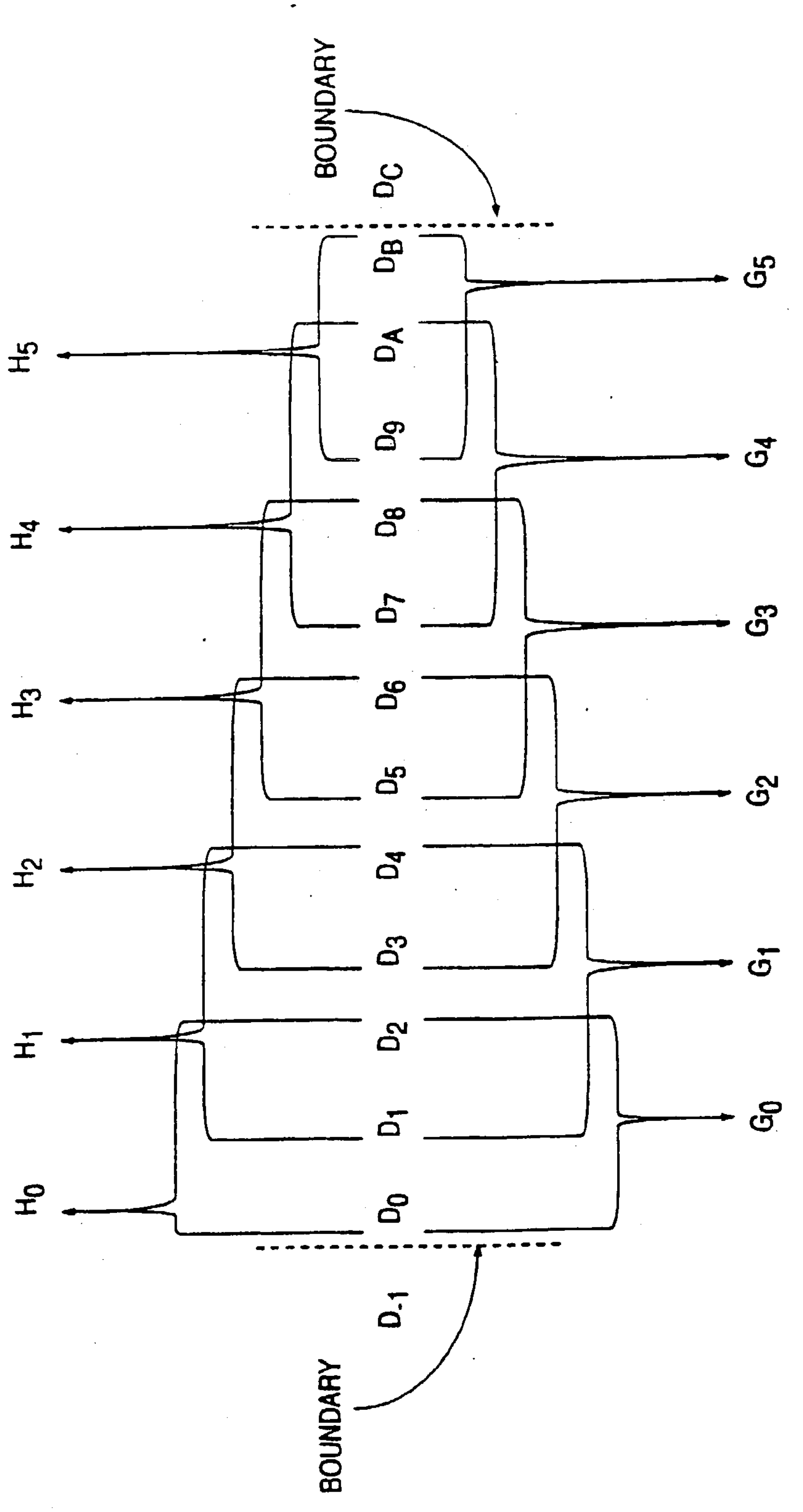

BOUNDARY
BOUNDARY
H0
H1
H2
H3
H4
H5
D-1
D0
D1
D2
D3
D4
D5
D6
D7
D8
D9
DA
DB
DC
G0
G1
G2
G3
G4
G5

Fig. 21

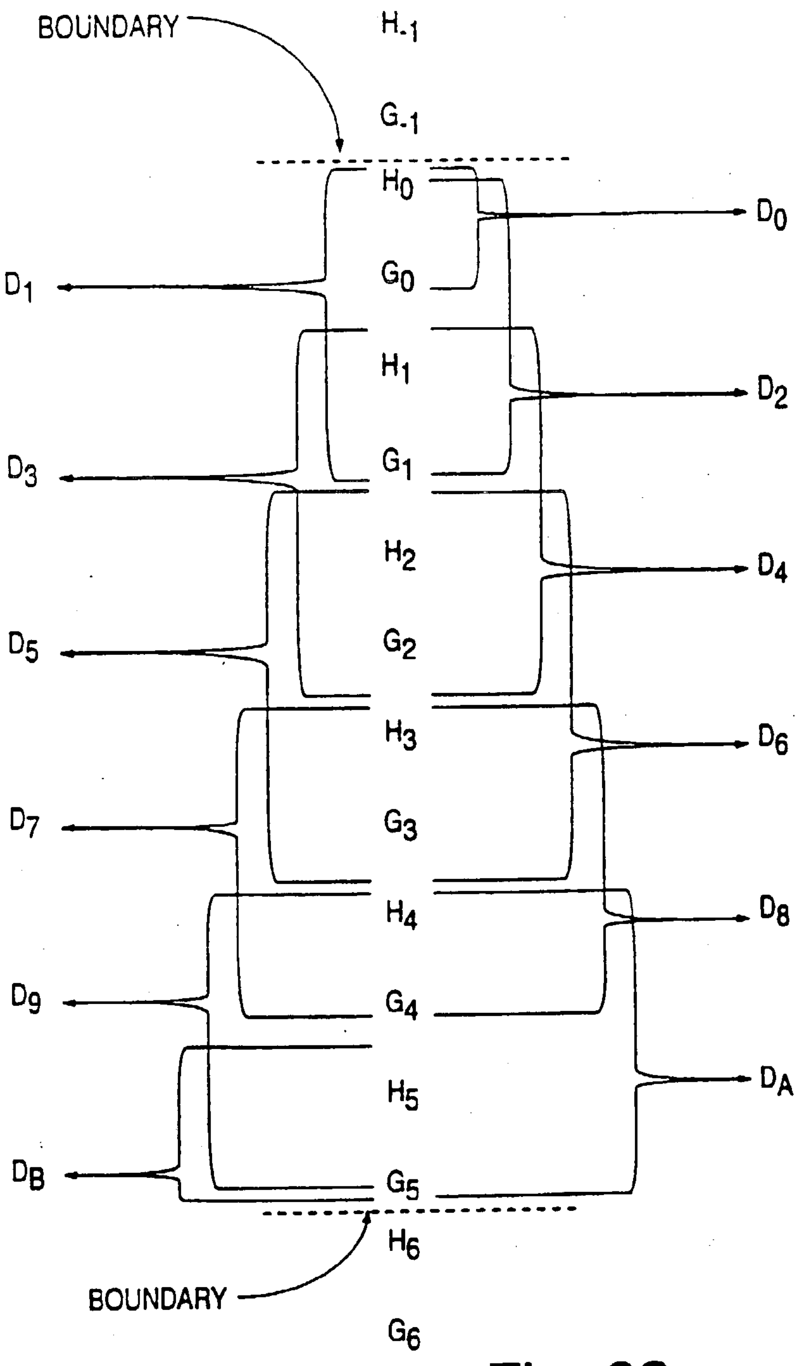

BOUNDARY
H-1
G-1
H0
G0
H1
G1
H2
G2
H3
G3
H4
G4
H5
G5
H6
G6
BOUNDARY
D0
D1
D2
D3
D4
D5
D6
D7
D8
D9
DA
DB

Fig. 22

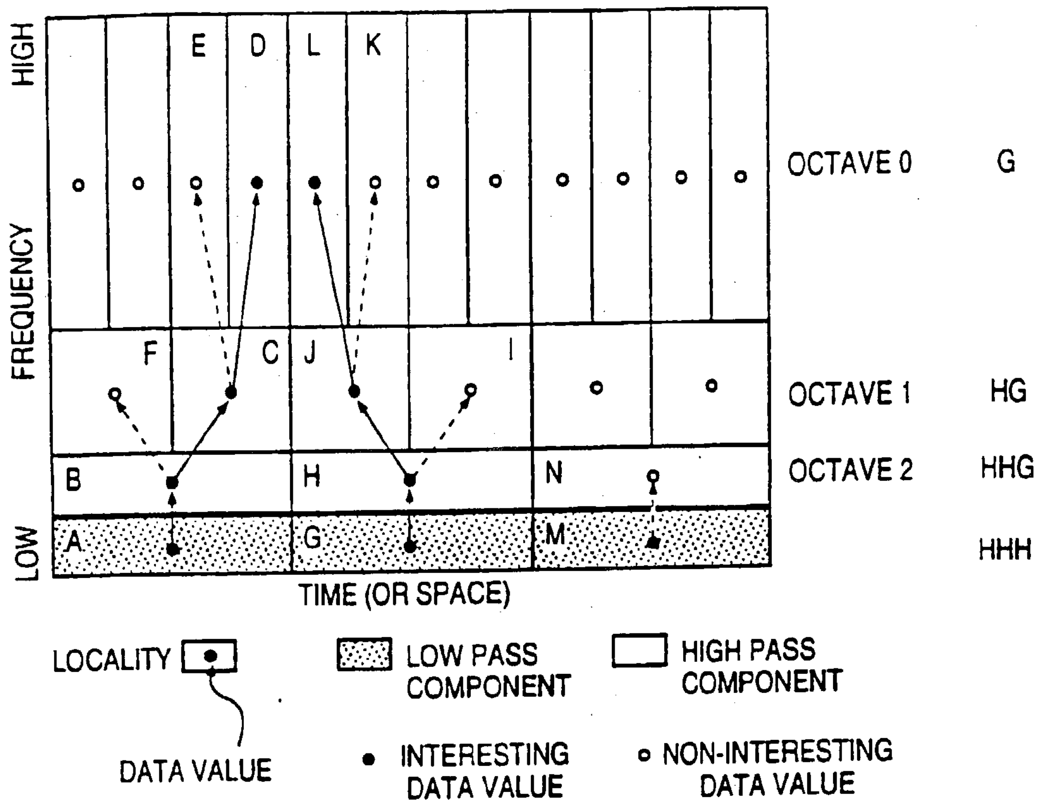

HIGH
FREQUENCY
LOW
E
D
L
K
F
C
J
I
B
H
N
A
G
M
OCTAVE 0
G
OCTAVE 1
HG
OCTAVE 2
HHG
HHH
TIME (OR SPACE)
LOCALITY
DATA VALUE
LOW PASS COMPONENT
HIGH PASS COMPONENT
INTERESTING DATA VALUE
NON-INTERESTING DATA VALUE

Fig. 23

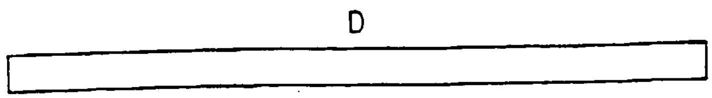


Fig. 24A

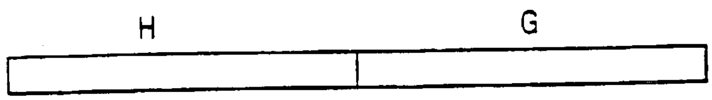


Fig. 24B

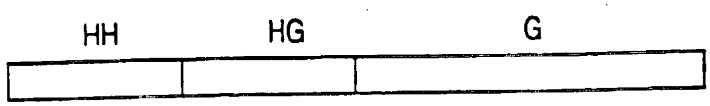


Fig. 24C

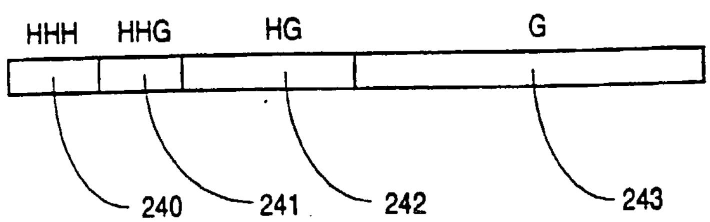


Fig. 24D

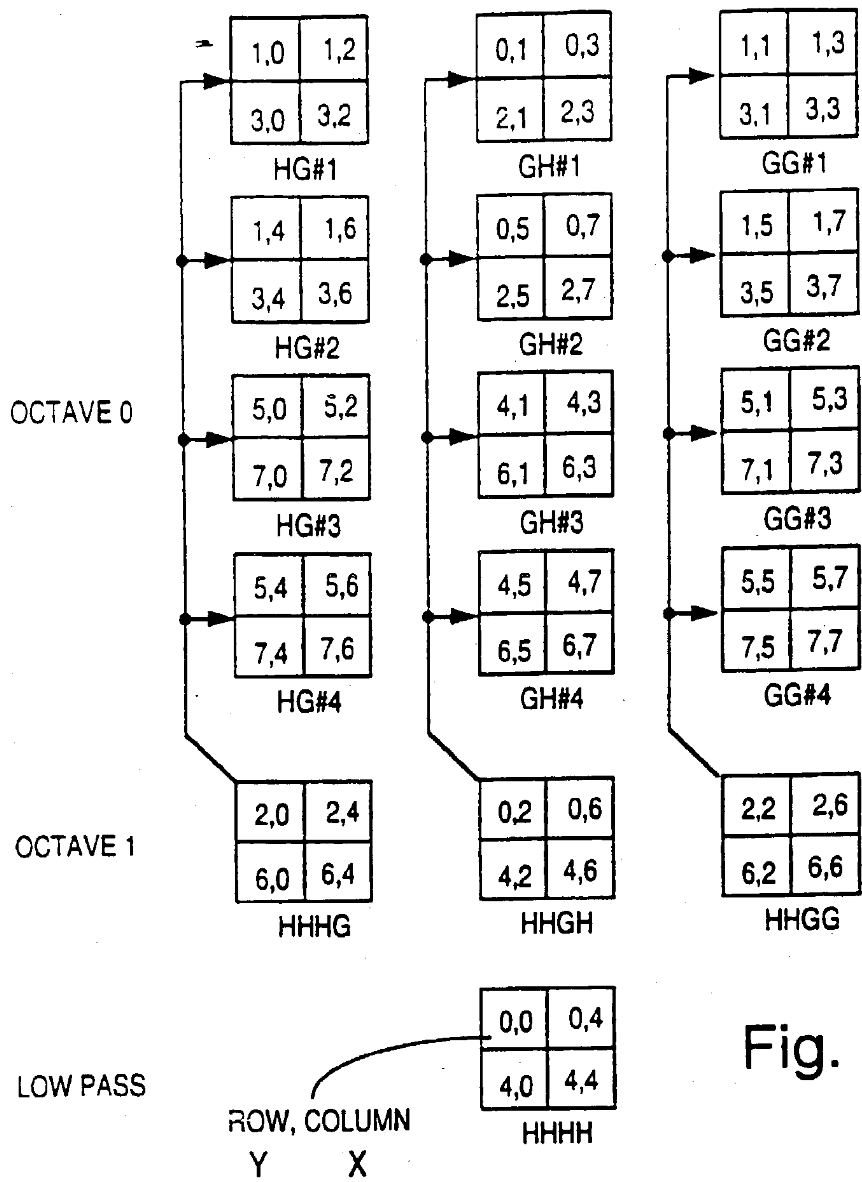
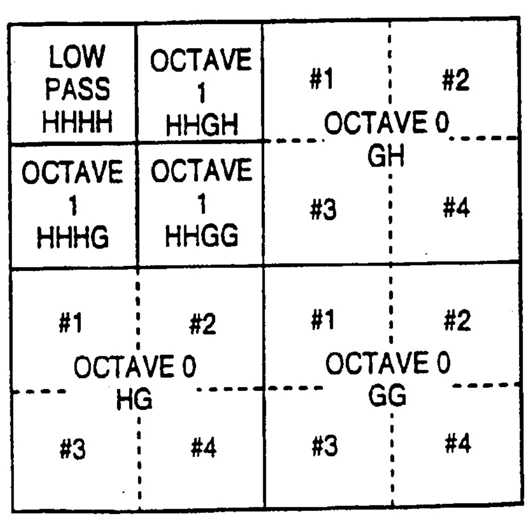


Fig. 25



PICTORIAL REPRESENTATION

Fig. 26

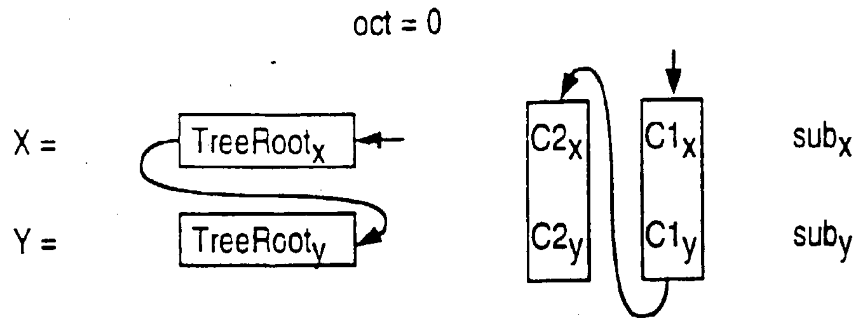


Fig. 27

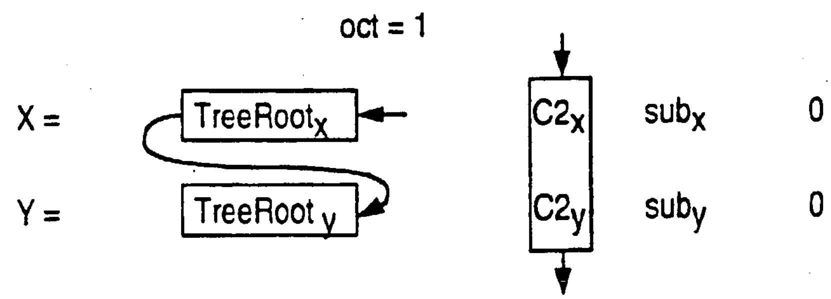


Fig. 28

	sub-band	sub_x	sub_y
low pass	HH	0	0
high pass	HG	0	1
	GH	1	0
	GG	1	1

Fig. 29

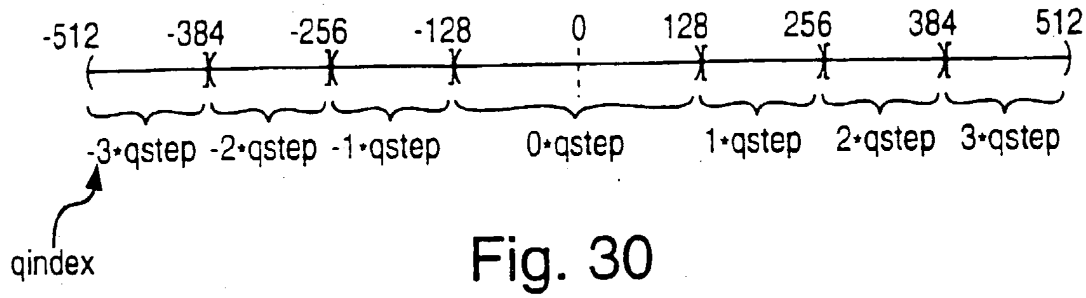

-512
-384
-256
-128
0
128
256
384
512
-3•qstep
-2•qstep
-1•qstep
0•qstep
1•qstep
2•qstep
3•qstep
qindex

Fig. 30

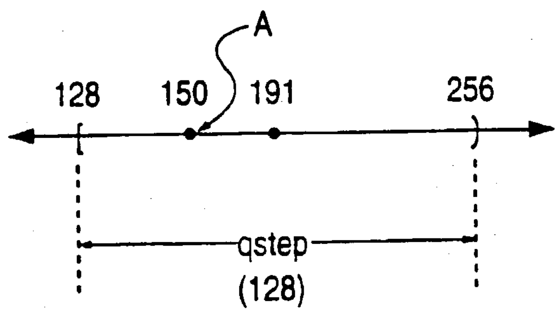

A
128
150
191
256
qstep
(128)

Fig. 31

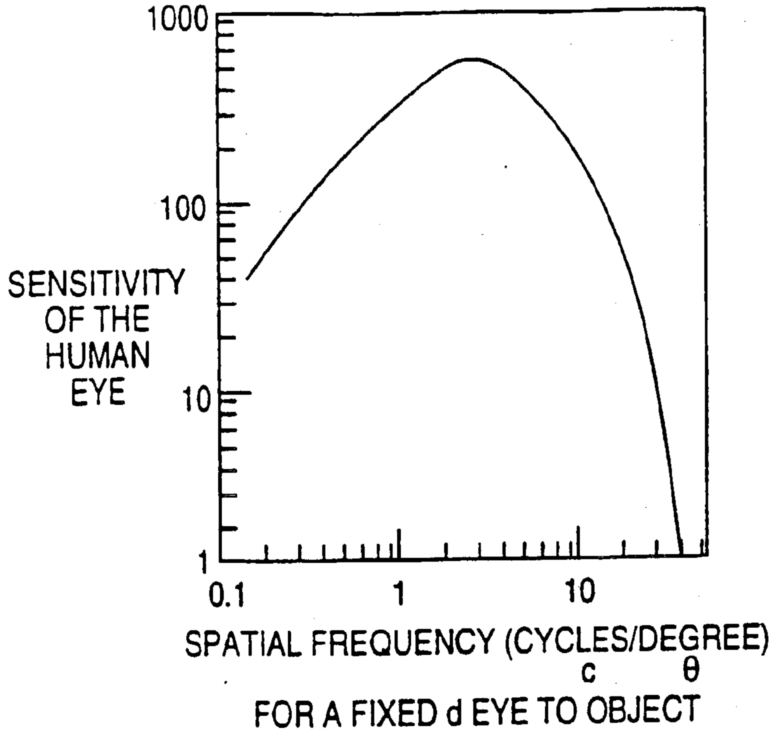

1000
100
10
1
SENSITIVITY
OF THE
HUMAN
EYE
0.1
1
10
SPATIAL FREQUENCY (CYCLES/DEGREE)
c θ
FOR A FIXED d EYE TO OBJECT

Fig. 32

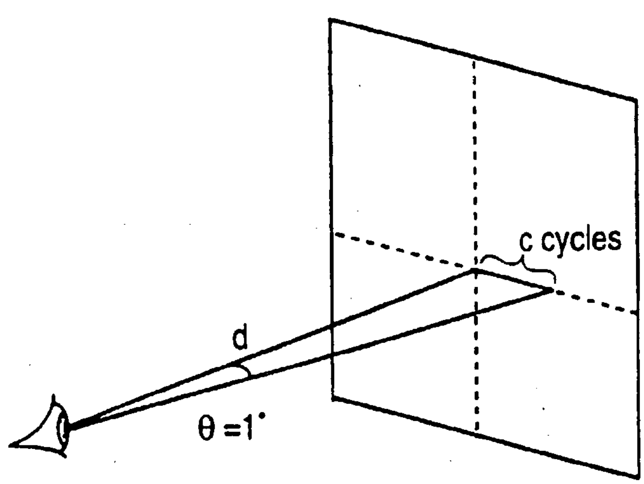

c cycles
d
θ =1°

Fig. 33

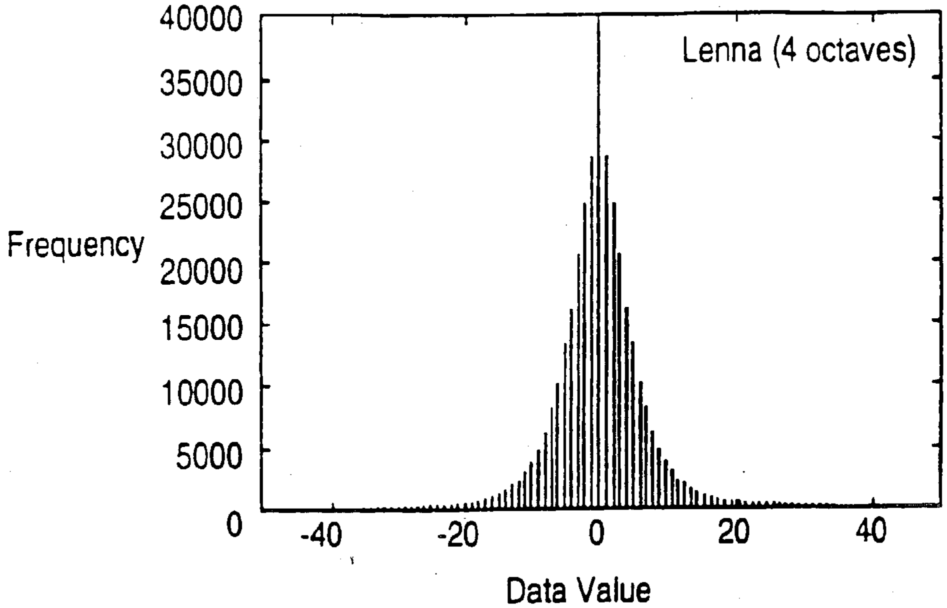

Lenna (4 octaves)
Frequency
40000
35000
30000
25000
20000
15000
10000
5000
0
-40
-20
0
20
40
Data Value

Fig. 34

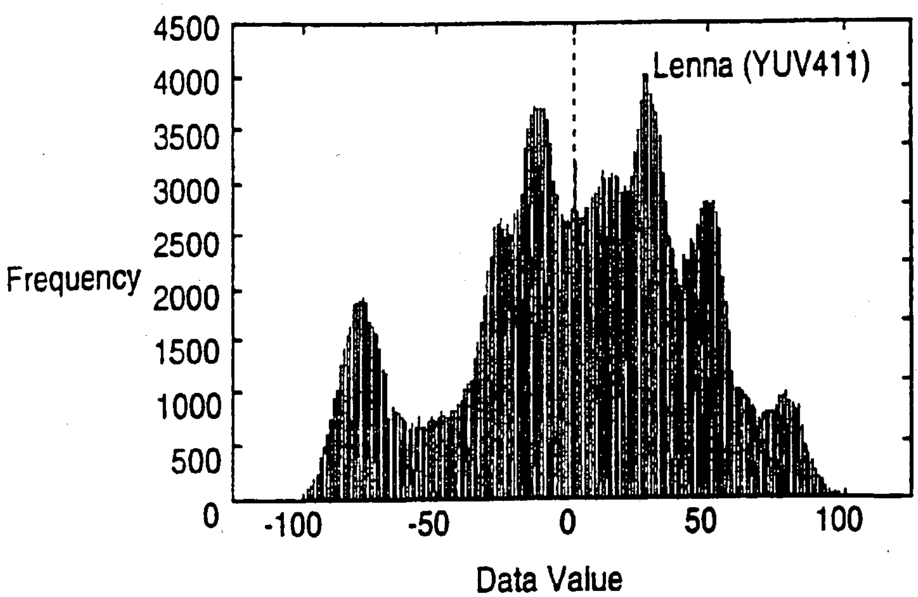

Lenna (YUV411)
Frequency
4500
4000
3500
3000
2500
2000
1500
1000
500
0
-100
-50
0
50
100
Data Value

Fig. 35

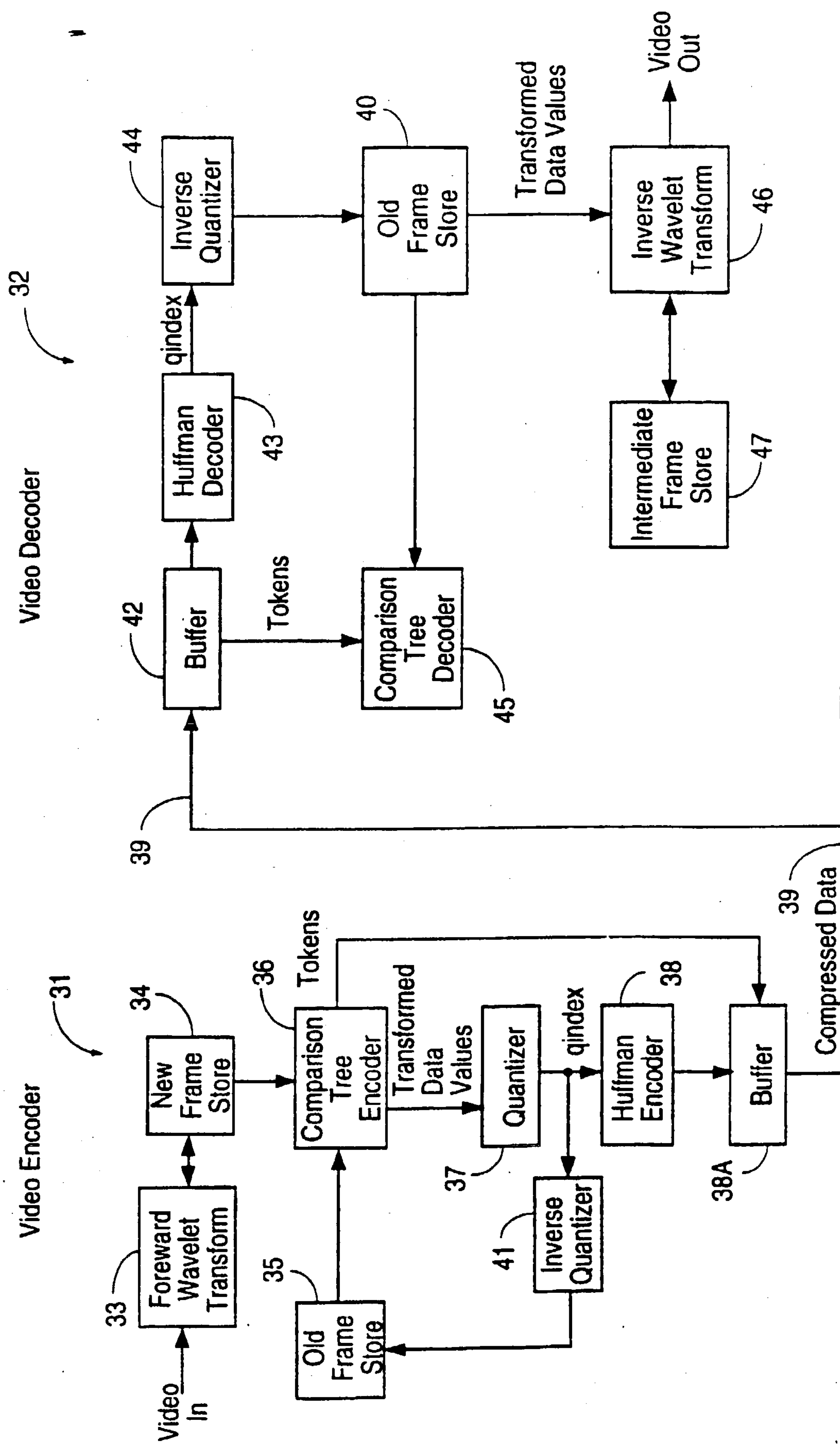

Video Encoder
31
Video In
Foreward Wavelet Transform
33
New Frame Store
34
Old Frame Store
35
Comparison Tree Encoder
36
Tokens
Transformed Data Values
Quantizer
37
qindex
Inverse Quantizer
41
Huffman Encoder
38
Buffer
38A
Compressed Data
39
Video Decoder
32
39
Buffer
42
Huffman Decoder
43
qindex
Inverse Quantizer
44
Tokens
Comparison Tree Decoder
45
Old Frame Store
40
Transformed Data Values
Inverse Wavelet Transform
46
Intermediate Frame Store
47
Video Out

Fig. 36

MODES OF VIDEO ENCODER AND VIDEO DECODER

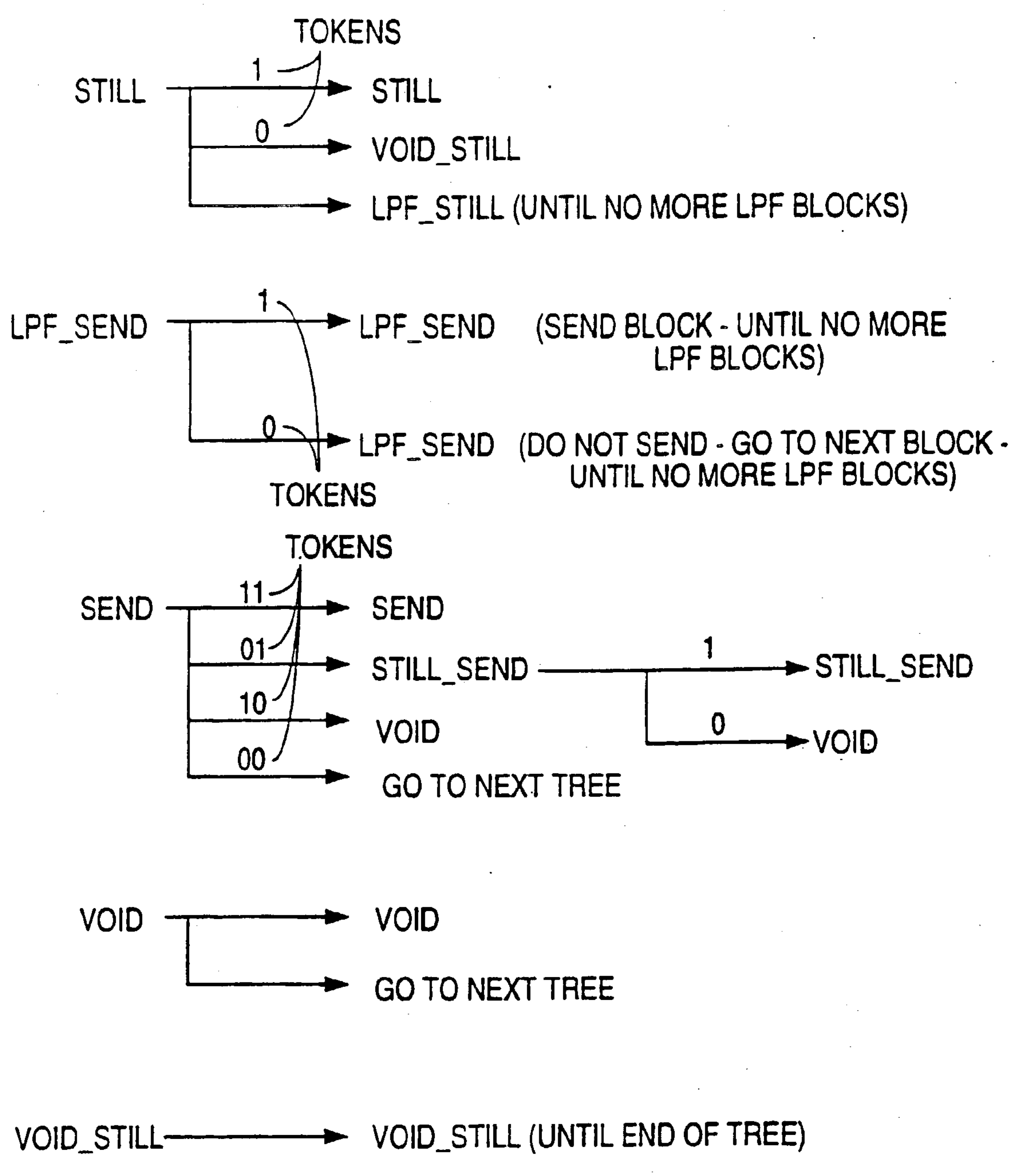


Fig. 37

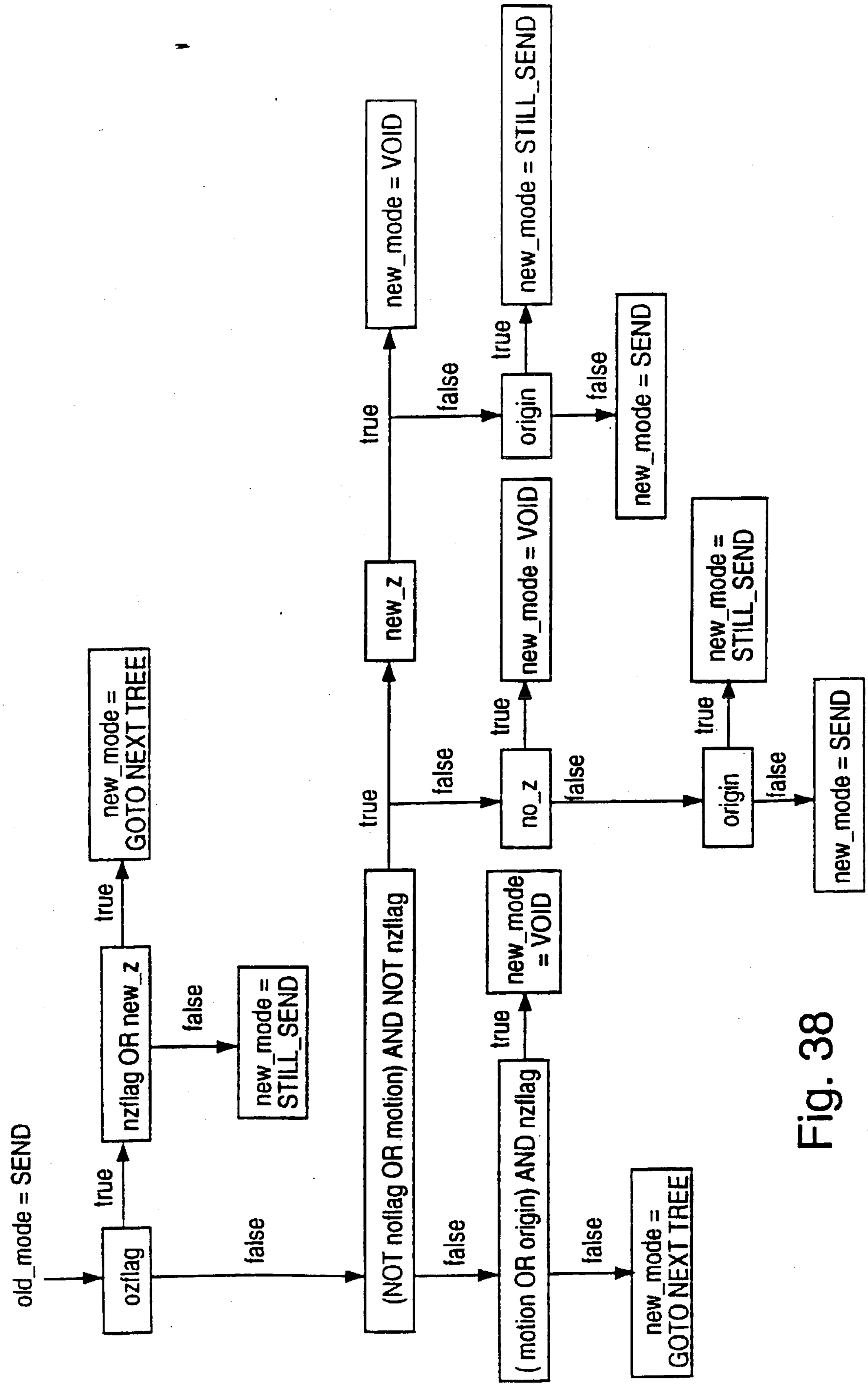

old_mode = SEND
ozflag
true
false
nzflag OR new_z
true
new_mode = GOTO NEXT TREE
false
new_mode = STILL_SEND
(NOT noflag OR motion) AND NOT nzflag
true
new_z
true
new_mode = VOID
false
origin
true
new_mode = STILL_SEND
false
new_mode = SEND
false
no_z
true
new_mode = VOID
false
origin
true
new_mode = STILL_SEND
false
new_mode = SEND
false
(motion OR origin) AND nzflag
true
new_mode = VOID
false
new_mode = GOTO NEXT TREE

Fig. 38

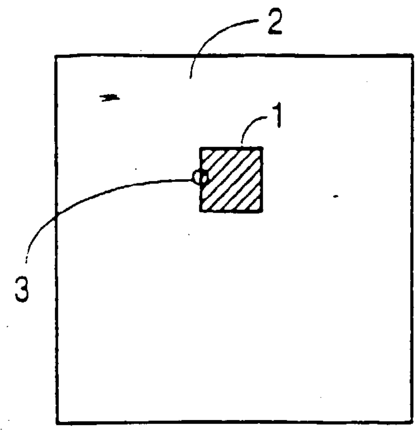


Fig. 39

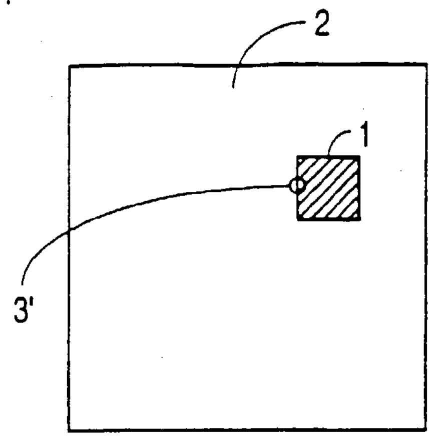


Fig. 40

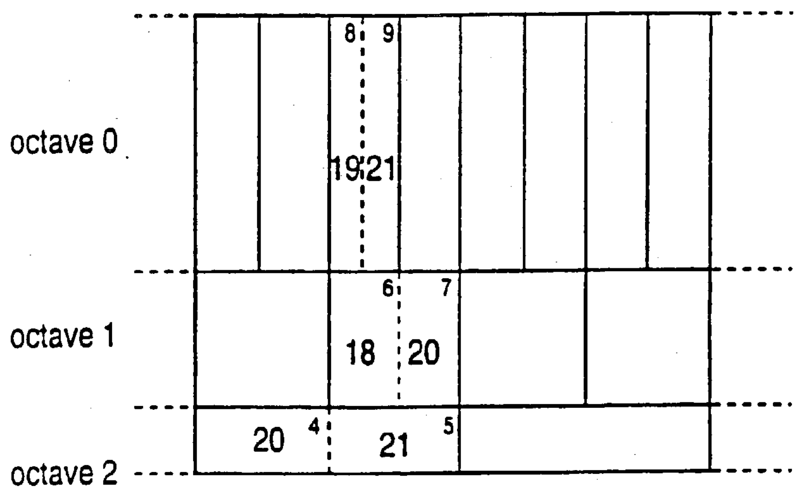


Fig. 41

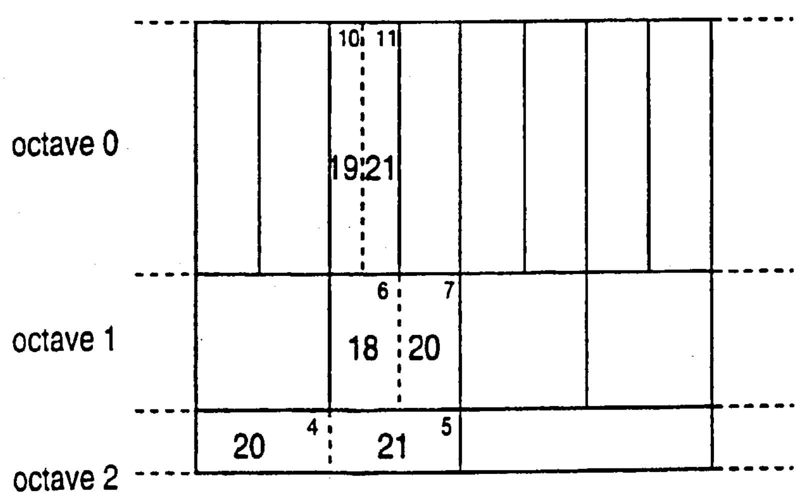


Fig. 42

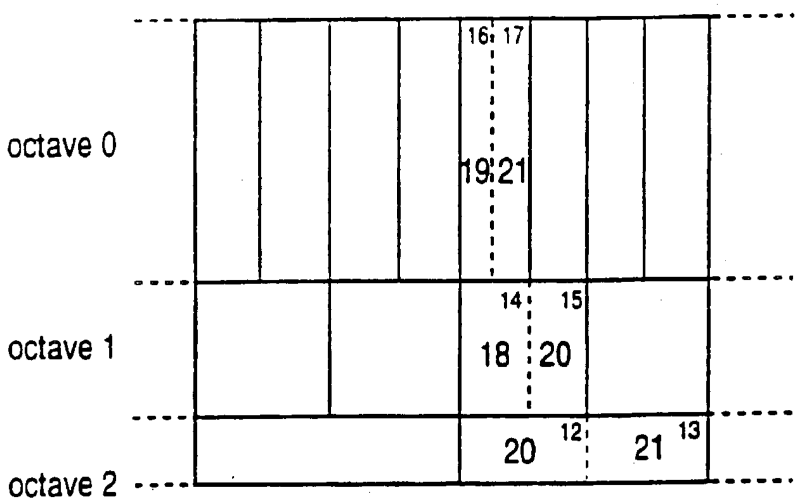


Fig. 43

VARIABLE - LENGTH TOKENS

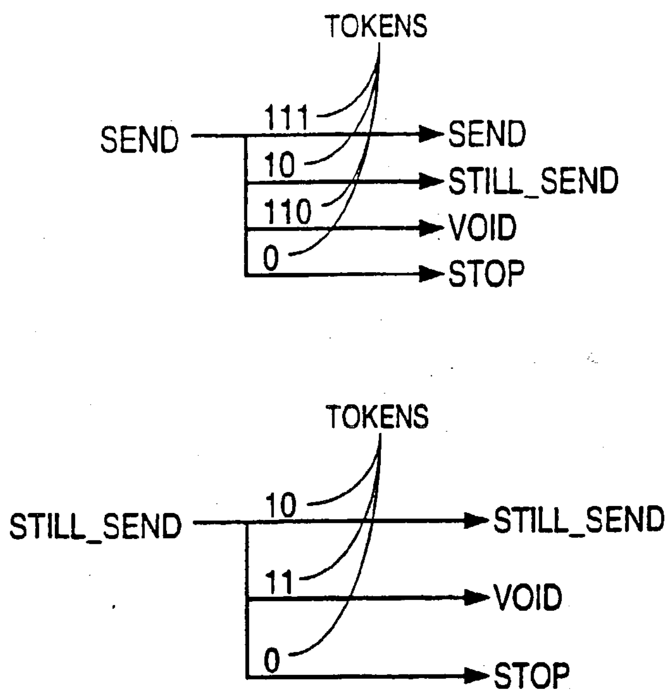


FIG. 44